Feasibility of Simulating the Java Programming Process

A thesis submitted in partial fulfillment of the requirements for
the degree of Bachelor of Science

by

W. B. Hueting

Student number:	851920848	
Course code:	IB9902	
Thesis committee:	chairperson (TBD),	Open University
	dr. ir. Hugo Jonker (supervisor),	Open University

Open Universiteit
de beste! www.ou.nl

Abstract

Fragmentation patterns on digital storage media could help forensic researchers to create file carvers and file- and operating system designers to improve their products. Fragmentation patterns are created by writing and deleting actions and relate to the software used. Researchers currently have to analyse real storage media over time to discover the semantics of these patterns. This is a time-consuming and privacy-invading method.

The goal of this research is to find a method to artificially create fragmentation patterns on storage media using user-behaviour workload. The method uses publicly available data on computer use instead of an underlying model to to generate input.

This research finds a method and implements a proof-of-concept that uses publicly available software development processes contained in a Git repository as input to simulate user behaviour workload.

The results show that small basic repositories can be used as input for a user-behaviour-workload-simulator, but more research on the characteristics of Git repositories is necessary to widen the pool of repositories that can be used as input.

Contents

Chapter 1

Introduction

Digital storage devices have become one of the most important tools of everyday life. In the information age we currently live in, user data has become one of the most important trade goods. Governments', businesses' and individuals' use of computers and smartphones increases everyday, depending more and more on digital storage devices. According to Eurostat [1] the percentage of people who use a computer or the internet on a daily basis has increased from 80 to 90 percent.

The information contained on these digital storage devices is not limited to pictures, text files and video's of a user. The process of writing to and deleting from the storage device creates patterns which could be used by digital forensic researchers to design and create file carvers, answering questions such as: what programs are used?, how long has it been used? and who has used it? File system and operating system designers could use this metadata for the optimisation of file- and operating systems.

A single storage device doesn't give nearly enough meta-information to derive such patterns. To discover these patterns and their semantics, data on a large amount of storage devices is necessary. In an ideal situation, storage devices of real-life users, who have been using them for years, are examined to derive usage patterns and information. Unfortunately, acquiring sufficient data to generalise usage patterns requires such a sizeable amount of used and usable storage devices, that acquisition becomes an obstacle to research. This is further exacerbated by privacy legislation, which requires consent for extracting usage patterns from each device.

A way to overcome these problems is to use synthetic experiments. Instead of acquiring real-

[1] https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Digital_economy_and_society_statistics_-_households_and_individuals

life data, in an experiment, use of a storage device is simulated. A problem with this method is whether the generated behavior is sufficiently approximating actual behaviour. Typically, such simulators rely on a mathematical basis to generate behaviour supposedly mimicking actual usage patterns. This pushes the question of realism to the mathematical basis.

Another approach is to simulate in full detail actual device usage. This has the clear advantage of approximating actual device usage quite reasonably, provided accurate data on device usage is available.

Sources of data on computer workload are free available on the internet. An example is the e-mail data set of Enron Corpus containing over half a million e-mails from 158 employees from the years prior to the collapse of the company. A great source of data but for the purpose of generating significant workload, drafting and sending e-mails is to lightweight.

We note that programming is a device-intensive task, involving typing, compilation / run cycles, debugging, using version control, etc. Moreover, thanks to the popularity of version control systems and open source, there is a treasure trove of data available on the construction of programs. More specifically, in recent years, GitHub has become a popular and free version control back-end for many smaller and larger open source projects. As such, it provides sufficient data for creating a realistic simulation of programming.

This project is a follow-up of the *Synthatic Fragmentation Experiments using WildFragSim* project by Robbert Noordzij [Noo19]. Noordzij's project resulted in a simulator that has a high level of realism in feeding input into the virtual machine. Noordzij achieved this by performing all processing on the host machine, and interacting with the virtual machine only via the regular mouse and keyboard interfaces. In this project, we aim to go extend his work into a simulation of user-behaviour workload.

Contributions. The main contributions of this thesis are:

- We posit that the data of a software development process, as captured by a version control system, provides sufficiently detailed and rich data to create a valuable simulation.
- We provide a practical exploration of using such data to simulate the a software development process.
- We provide a proof-of-concept implementation for replaying software development projects based on data stored in a version control system.
  Our proof-of-concept tool supports directly importing a project from GitHub, branching and merging in Git, replaying development in the Eclipse IDE, and committing. Support for continuous integration / continuous deployment is beyond the scope of this proof-of-concept tool.

This thesis finds a method to generate user behavior workload by replaying the programming process mimicking a programmer. We make use of GitHub, a website containing lots of programming projects with a detailed development history. The core idea is to let the simulator replay the entire development history i.e. starting an IDE, typing of code and adding the newest version to the version control system.

This research does not involve creating user profiles to be realistic. However, to study the workings of the method, a simple programmer profile will be established. To scope the research, it focuses on replaying Java projects with Eclipse as the IDE.

Chapter 2

Background

2.1. Version Control Systems

Version control systems (VCS) are a class of software tools that assist a software team manage changes in source code over time. Every change to the code between versions(commit) is saved as a file containing changes relative to its parent (deltafile) and saved in a special database. Version control systems are further dividable into Central Version Control Systems (CVCS) and Distributed Version Control Systems (DVCS).

In CVCSs there is one truth, one central location of the entire project's history and interacting with the history requires a network connection. An example of a popular CVCS is Subversion (SVN) [1]. As opposed to CVCSs, DVCSs do not rely on a single place for the full history of the project. Everybody's working copy of the code is also a repository that can contain the full history of the project. This means that interacting with the repository only happens locally and no network connection is necessary. An example of a popular Distributed Version Control System is Git [2].

To allow sharing of- and working locally with the entire history, Git has to think differently about saving changes. SVN thinks of the information they keep as a set of files and the changes made to each file over time (delta file).

Fundamentally, Git does save delta files[3] [4], but on a conceptual level thinks completely different about changes. Every time you make a commit of your project in Git, it takes a snap-

[1] https://subversion.apache.org/
[2] https://git-scm.com/
[3] http://git-scm.com/book/en/Git-Internals-Packfiles
[4] http://git-scm.com/book/en/Git-Internals-Git-References

shot of what all your files look like at that moment and stores a reference to that snapshot. For efficiency, Git doesn't store unchanged files again, but instead adds a reference to the identical file it has already stored.

Thinking about data this way gained some benefits and are the reasons Git is so popular.

- You are always working in a local repository, no network connection is necessary to browse the history.
- Every snapshot is checksummed and referred to by that checksum.
- Branching is very lightweight.

A Git project consists of three main sections (figure 1); the working directory, staging area and the repository. The working tree is a version of a software project loaded in memory to use and modify. The staging area is a file containing information about what will go into your next commit. The repository is where Git stores the metadata and database for the project. The repository is what is copied when a project is cloned to another computer.

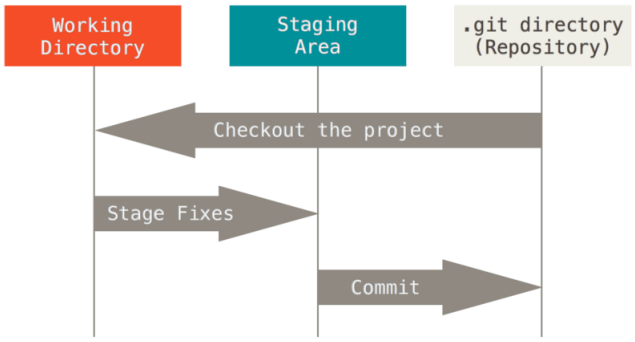


Figure 2.1: Sections of Git project

2.1.1. REPOSITORY

When a commit is made, Git checksums the subdirectories placed in the staging area and creates a tree object containing these checksums. Then a commit object containing metadata, such as the author and the committer, and a pointer to the tree-object is made. Committing three files results in five objects in the repository; a commit object, a tree object and three "blobs" containing the contents of the staged files. Because everything is checksummed and referred to it by that checksum, it is impossible to change files without git knowing about it. This is one of pillars of Git and guarantees the integrity of files. Referencing to checksums also has the advantage when a file is staged that is the same as a file in the repository. Git will not make a new blob for this file but references to the already present blob in the repository.

This will keep memory use to a minimum.

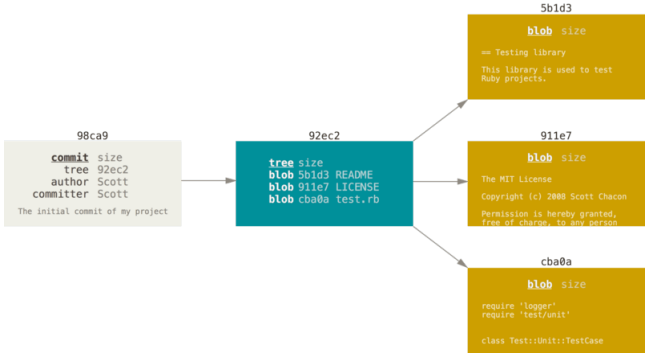
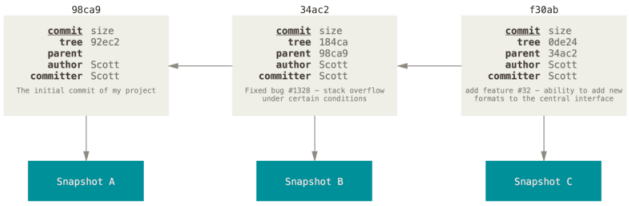


Figure 2.2: Commit; white: commit object, blue: tree object, yellow: blobs

After making changes to the files and committing again. The created commit object will contain the checksum of the previous commit as a reference. Doing this a couple of times will create a tree of commit objects with the initial commit as its root.

Git also creates a pointer called a branch. This pointer points to the commit(or version of project) you are currently looking at. This initial pointer is named "master" by default.



2.1.2. BRANCHING AND MERGING

The concept of **branching** is to divert from the main line to work on the files without messing with the main line and allows multiple users to do different work on the same project. Nearly all VCSs support branching but is often an expensive process requiring to make a full copy of the source code, which can take a while with large projects. Creating a branch in Git is a very lightweight process. All it takes is adding a pointer pointing to the commit diverting from. This makes branching in Git nearly instantaneous.

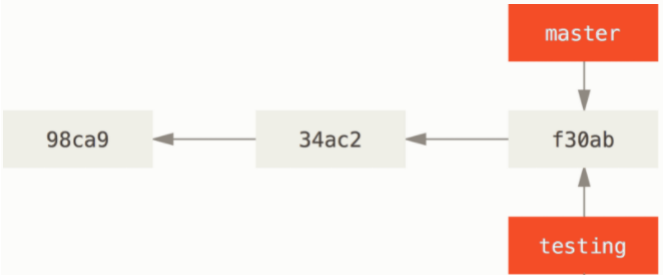


Figure 2.3: After the creation of branch "testing", two branches exist in the repository.

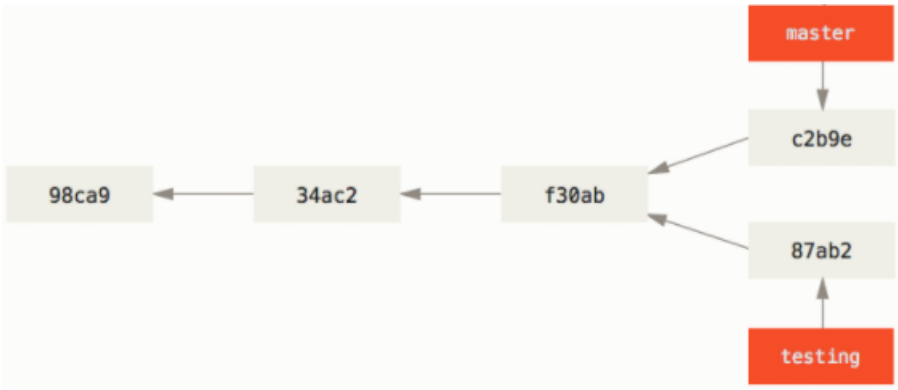


Figure 2.4: After both branches have committed a different commit. Two commit objects reference the same parent commit.

Once work has completed on a branch, it can be added to another branch. This process is called **merging**. Merging comes in two flavours; fast-forward and merge-commit.

Fast-forward merging is used when merging a branch with a branch that points to an ancestor. In this case, the branch pointing to the ancestor is moved or fast-forwarded to the same commit as the newest branch is pointing to.

Merge-commit is used when merging a branch with a branch that is pointing to a commit

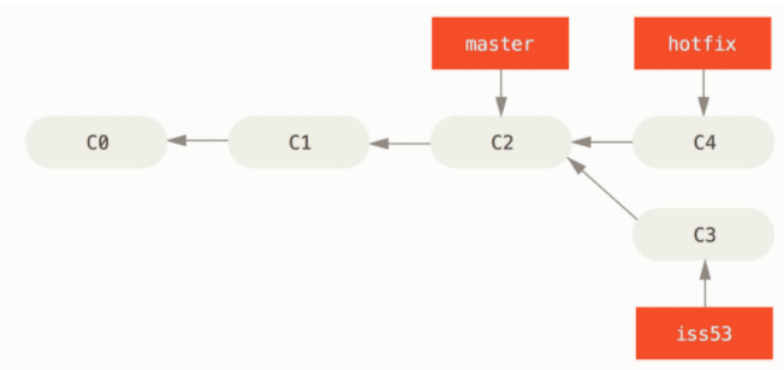


Figure 2.5: Before fast-forward merging c2 and c4. c2 is ancestor of c4.

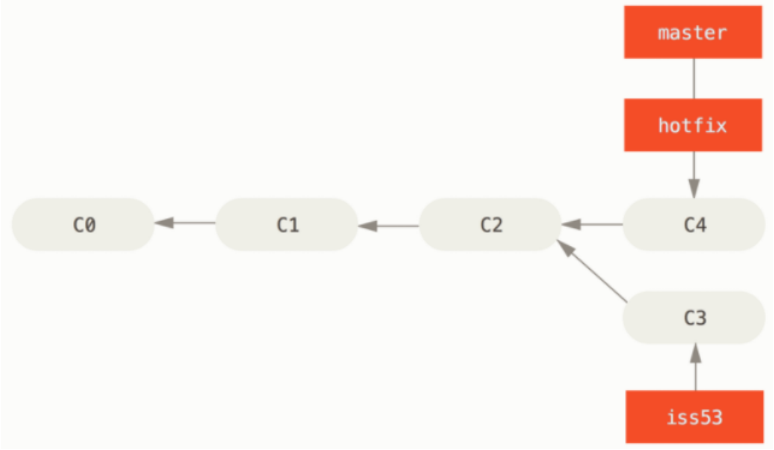


Figure 2.6: After fast-forward merging. Branch "master" is fast-forwarded to branch "hotfix"

that is not an ancestor, but has a common ancestor. This means that the changes have to be combined and can result in a merge-conflict. A merge-conflict arises when the two branches contain different changes of the same object referred to in the common ancestor. When this happens the committer can make a choice which version to use or change the files before merging again. If the merge-conflicts are resolved or no merge-conflicts arise, a new commit containing a new snapshot consisting of the combinations of the two snapshots pointed to by the two branches is created.

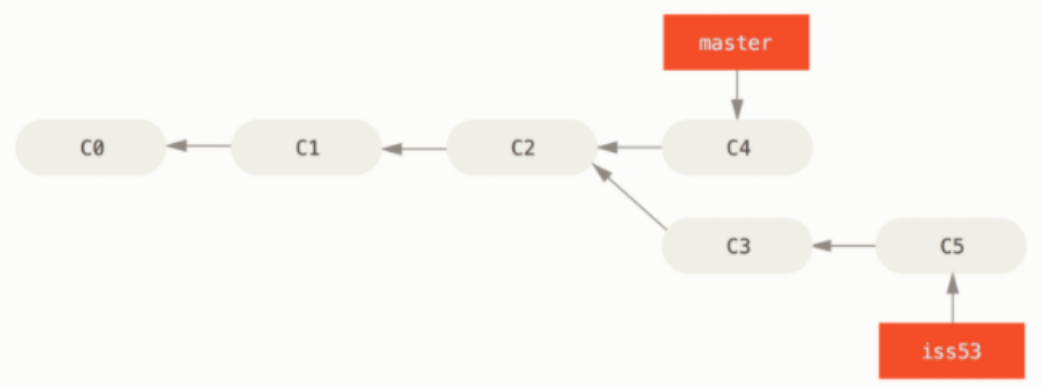


Figure 2.7: Before merge-commit. Merging c4 and c5 with common ancestor c2.

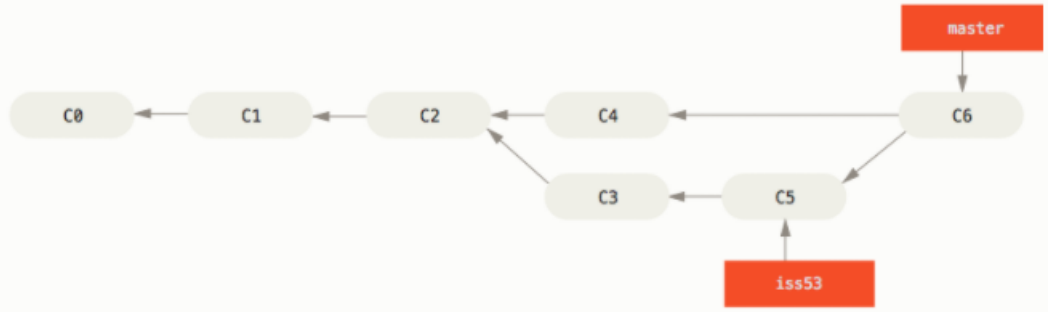


Figure 2.8: After merge-commit. New commit object c6 containing new snapshot that is the combination of c4 and c5.

2.1.3. DIFF CHUNKS

We are interested in the differences between commits. Git is capable of determining the changes between commits and blobs and shows these changes as diff chunks. A diff chunk contains the actual changes between blobs and some metadata. For changes between commits (patch), the output shows metadata of the commit and one or more diff chunks.

The metadata of a diff chunk shows information such as; if a file has been created, deleted, renamed, changed or merged. If it is adapted and the changes are textual. It shows what lines are adapted. The diff chunk is the basis for the input for the simulator. In figure9, an example of a commit is given including a single diff chunk. This patch is the result of running the command: git log –patch a400f600ad8e8146717683427a2517b0fc819eb2 31c1c94dc35066c6327535b984b55961701dc396 in this repository [5].

```
commit 31c1c94dc35066c6327535b984b55961701dc396
Author: Shyam Thiagarajan <ShyamW@users.noreply.github.com>
Date:   Sat Oct 8 13:44:26 2016 -0400

    Update HeapSort.java

diff --git a/HeapSort.java b/HeapSort.java
index ed3a07d..7dd420a 100644
--- a/HeapSort.java
+++ b/HeapSort.java
@@ -1,7 +1,7 @@
 import java.util.Scanner;

 /**
- * Heap Sort Algorithm.
+ * Heap Sort Algorithm. Implements MinHeap
  *
  */
 public class HeapSort {
```

Figure 2.9: Changes of commit with diff chunk

The first three lines are metadata of the commit followed by the commit message. The diff chunk starts on the line beginning with diff and shows what blobs it concerns. Between the "@@" are the lines affected followed by the textual changes. In this commit the line "Heap Sort Algorithm" in the old file is changed to "Heap Sort Algorithm. Implements MinHeap".

[5] https://github.com/TheAlgorithms/Java.git

2.2. AUTOMATING USER INTERFACE INTERACTION

There are various ways to automate user interface action. Roughly speaking, either the user interface is entirely processed by the automation, or there is a specific API that opens up the user interface to scripting.

Microsoft provides such an API in the .NET framework, the UIAutomation framework.[6]. This API addresses the needs of assistive technology products and automated tests frameworks. It provides programmatic access to information about the user interface (UI). This enables products like screenreaders and automated test scripts to interact with the UI.

UIAutomation discloses all the elements in the UI in a tree structure with the desktop as its root element. The root element for example has application windows as its children and those application windows have buttons and menu-items as its children.

The framework exposes every element in the UI as an AutomationElement. An AutomationElement contains all properties of the UI element e.g. an AutomationID, a name property, a controltype (e.g. button or checkbox) and controlpatterns (e.g. click a button or check a checkbox).

Writing scripts to interact with these AutomationElements requires the discovery of the properties of AutomationElements in already existing applications. With these properties, the framework is able to find the elements in the UI. To discover these properties Microsoft made Accessibility Insights[7] available. With this program you can discover the properties of UI elements.

[6] https://docs.microsoft.com/en-us/dotnet/framework/ui-automation
[7] https://accessibilityinsights.io/

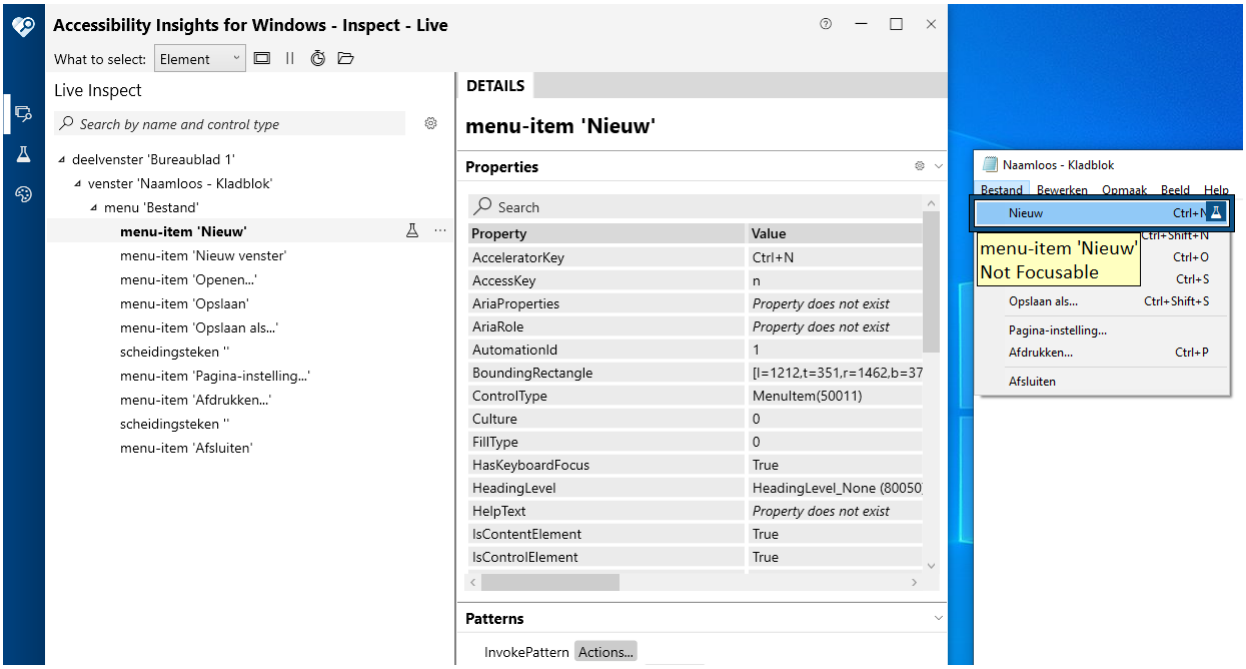


Figure 2.10: Screenshot of AccessibilityInsights with the focus on the menu-item "Nieuw"

In Figure 2.10 we have put the focus on the "Nieuw" menu-item from the drop-down list under "Bestand" in the Notepad application. On the left we can see the branch from root-element "Bureaublad 1" to the menu-item we have our focus on. In the middle we find a selection of properties of the menu-item (e.g. AutomationID and ControlType). With its place in the UI-tree in combination with its properties, we can uniquely identify the correct UI element within the UI. On the bottom, we can see what actions are allowed on the element. In this case, the menu-item can be invoked via the InvokePattern. When we have found the correct UI element, we can can also get its location on the screen using x- and y-coordinates. These coordinated can be used to guide the mouse to a point on the screen.

Chapter 3

Related work

Generating realistic computer workload has been a subject of study for many years. One of the many types of realistic workload is user behaviour. User behaviour encapsulates the order of user interactions at the higher levels [HK99, Fis01]. These interactions are, for example, the commands the user runs on the system or his habits in using a particular feature e.g. usage of keyboard, mouse, touch and speech.

Workload generation can be done by copying real data or by data generated by a model. A model is a rule-set derived from (typically) real data, which produces synthetic data. As synthetic data is produced by a rule-set containing generalisations and assumptions based on real data [HHK00], it can, at best, provide an approximation to real-world data. This makes the use of synthetic data controversial and frowned upon [CMCU04, Gan95].

In literature, a very large amount of work exists on the characterisation, modeling and simulation of workload generators (e.g. to peer-2-peer file sharing [GDS+03], malicious and unwanted traffic [BV14], live streaming media [VAM+02], YouTube [AS10], web proxy caches [BW02], IDS-training [GVUK06]). Unfortunately, we cannot say the same thing about generation of realistic workload. On the contrary, several works whose title alludes to producing a realistic workload still use model-based workload generation [AAM04, HCSJ04]. Other works mention the desire to use realistic data, but were unable to due to a variety of reasons, e.g., privacy constraints, or insufficient amount of realistic data [CMCU04, GVUK06, JSL+05].

However, for certain specific categories of work, sufficient data has become publicly available. In particular, user data on software development is publicly available via public version control systems (VCS), such as GitHub and GitLab. These VCSs contain a large quantity of publicly available software projects and their entire development history. This readily available development history can be used as a basis for a workload generator. This requires

parsing the history of these software projects for re-playing them. While there are several tools to parse these histories, they all focus on source code analytics [ZW04, FPG03].

Finally, Noordzij worked on a spiritual predecessor of this project [Noo19]. His approach was to keep the simulated machine free from any processing. To that end, his implementation sends scan codes via the virtual machine's keyboard interface. Output from the virtual machine is processed via screenshots. This is rather cumbersome and fragile: many factors can cause screenshots to subtly change, bringing the simulation to a halt. The clear benefit is that all processing of simulator output happens outside the simulation, and thus does not impact the simulation. An interface within the simulation could greatly simplify and stabilise this aspect, but comes at the expense of realism. We will investigate alternative approaches to processing simulator output.

Chapter 4

Automatically replaying a git repository

We want to give the simulator a large workload to see what the effects of the write and delete actions are on the storage medium. Our approach is based on simulating input taken from a Git repository. This means that the Git repository needs to be downloaded and parsed to provide input to the simulation. To ensure the experiment remains clean, we need to distinguish between the computational effort of providing input to the simulation and the simulation itself. In case the goal is to maximize the workload, both tasks can be executed in the same environment. If the goal is to approximate the workload of a software development process, the simulation environment should not be "polluted" by the workload needed to provide input to the simulation.

The latter is the goal of this study. We want to approximate the development process as a programmer would: using an IDE and a VCS to code, compile and commit.

As a consequence, we need an environment where the development process is simulated and an environment where the simulator is controlled by downloading, interpreting and converting the original repository. To make the experiment and production of data more scalable and practical, a virtual machine (VM) is used to run the operating system where the simulation takes place. The use of a VM also adds the advantage that allows us to easily revert to an earlier stage of simulation or to the initial starting conditions using snapshots.

Scope and limitations. This research main focus is to correctly replay a the development process contained in a Git repository. To further scope the research, only Java code is replayed and is executed on Eclipse as the IDE. All other non-Java files in the repository are copied from the original to the simulated repository. The generated projects are committed to a local repository. We do not download and add external non-default libraries as there is no default

method to do this for Java. Specific solutions do exist, but there is no common standard used by all projects (in contrast to e.g. CPAN for Perl, CTAN for LaTeX, and pyPI for Python). As a consequence of not adding external libraries, we do not compile the replayed code. Because only projects using the default libraries will compile and we would therefor need to distinguish default-library projects from non-default-library projects. We also do not support Maven of Gradle based projects. Finally, the simulator is to support UTF8 character encoding where possible. However, this proof-of-concept focuses on realism by simulating typing via the keyboard interface. This introduces a complication for UTF8 characters not present on the current keyboard layout, which we leave to future work.

4.1. SIMULATION PROCEDURE

To replay the development history of a repository, the commits contained in the repository need to be replayed as a programmer would. At first we need to download and parse the repository, start the VM and initialize the IDE. When the environment is ready, the commits can be replayed in chronological order. Each commit following the general programming procedure: (code -> compile)$^{1..n}$ -> commit. The code step includes creation, deletion, update or rename/relocation of code.

The commits in the repository are structured as a directed acyclic graph which could contain multiple parallel development lines. These different development lines are represented by different workspaces in the IDE. To make this work, we need to accommodate branching and merging. Each time a branch is encountered, a new workspace and repository has to be created. Each time a merge is encountered, the concerning workspaces and repositories need to be merged. A merge is encountered as a commit so the merge action is contained in the code step. A branch is encountered as a commit having multiple children. As a consequence of replaying multiple parallel development lines in chronological order, we sometimes have to change the workspace we are working in. The change of workspace needs to be done before the commit is replayed.

Combining the above steps results in the simulation procedure represented in appendix A.

Each step of the procedure has its own challenges and solutions and are outlined in the following sections.

4.2. REPOSITORY INTERACTION

To use the data confined in the repository, we need to discover and parse the data into user interface interactions. Git offers a rich toolset for the CLI to download and interact with a repository. The actions to discover and parse data are executed on the host. As such, there is no need for realism, so all actions can be executed via Powershell scripts.

We will use publicly available Git repositories as input for the simulation. Once the repository is cloned, we extract the entire commit history (using Git's logging tools) from the repository as a text file. This text file contains all commits with metadata in chronological order. The contained diff chunks together represent the entire development history.

This file is then parsed into a directed acyclic graph of separate commits, each with their own metadata (name+email of the committer, reference to parent commit, Boolean indicating whether this was a merge).

Where to parse the development history? There are two ways to divide parsing development history between host and simulation:

1. The host parses each commit and feeds the simulation only what is currently needed;
2. The helper program parses each commit inside the virtual machine.

Letting the host parse the commit seems desirable. However, it requires host and simulation to be synchronised with respect to the current state of the virtual machine. Parsing commits on the virtual machine does not such strict synchronisation.

Both approaches increase the computational efforts of the virtual machine. The former via a more complicated and intensive way of communicating, the latter via the extra parsing jobs. We choose to perform parsing on the virtual machine, as it significantly reduces the requirement for synchronisation.

4.3. INTERACTION WITH USER INTERFACE

As previously stated, a program on one operating system needs to control the simulation on another operating system. Our preference is for a way where the controlling program inputs the necessary actions as an actual user would, without interfering with the simulation. This means interpreting the UI by finding and clicking the correct elements, and coding with a realistic typing rythm.

Interpretation of the UI can be done via image recognition, as shown by Noordzij [Noo19]. He managed to do this by supplying the program with images to find on a snapshot of the

to-be-interpreted UI. Unfortunately, he ran into the problem that the program was unable to locate the correct elements, when the visual representation of the elements changed, for example, by a software update. The program needed to be resupplied with new images.

If we want to use this method, we have to supply the program with the correct images to find throughout the entire software development process. This is hard, as there may be user-defined UI elements (e.g., labels, as shown in Figure 4.1). This means that images for UI image recognition cannot be created in advance, but need to be created on the fly. Moreover, it is plausible that multiple representations of a single element (e.g., a function- or method-name) exist within the UI (as shown in Figure 4.2). Distinguishing between multiple such instances requires context of each instance. Since functions (in general) can be arbitrarily nested, there is no guarantee how much context suffices. Because of these disadvantages, image recognition is not a viable approach to use for interaction with the user interface.S

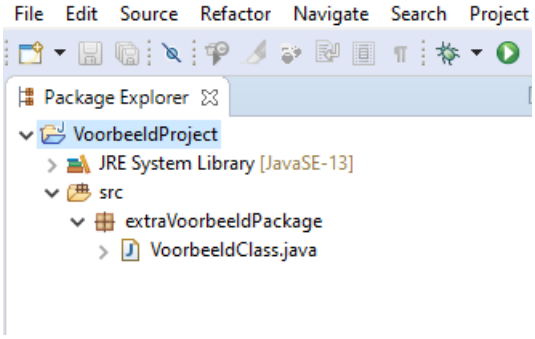


Figure 4.1: User-defined UI elements "VoorbeeldProject", "extraVoorbeeldPackage" and "VoorbeeldClass.java".

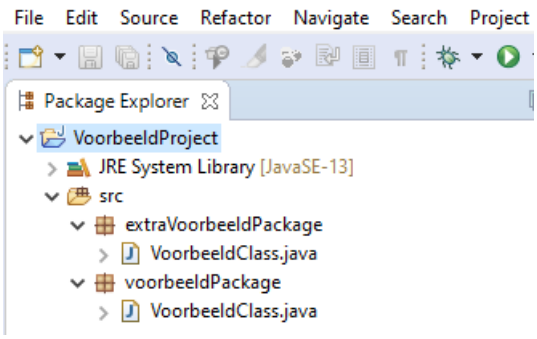


Figure 4.2: Two user-defined UI elements "VoorbeeldClass.java" exist within the UI.

We need a context-aware approach to interpret the VM's user interface without relying on its visual representation. Within the VM, context is available. UIAutomation can provide us with the context necessary to find the correct UI elements. As such, it stands to reason to use a helper program located on the VM. However, using a helper program incurs a disadvantage: it influences the result of the simulation, since running the helper program adds to the workload. To gauge the impact of a helper program, we performed an experiment to measure memory- and disk-use of such a helper program.

Experiment: validation of helper program. To determine its impact, a small helper program was developed[1] that retyped publicly available books[2] as textfiles. The program needs

[1] https://github.com/WHueting/HelperProgramTest
[2] https://www.gutenberg.org/

to find and invoke the correct user interface elements via UIAutomation.

To retype the book, first notepad has to be started. This is done by finding the shortcut for notepad on the desktop. After invoking the shortcut, notepad is started with an empty page ready for input. The program then start retyping the entire book. After its completion, the book has to be saved. The save button on the notepad window has to be found and invoked. This results in a save-window to enter a name for the created textfile. The corresponding name is typed in the correct textbox and is confirmed by the "return" key. All the typing is done by sending corresponding keystrokes to the active window or application.

Using Visual Studio's diagnostic tools we determined that the program used 18 MB of memory and on average 2% of the CPU with peaks up to 4%. To determine if the resource usage increased when input increases, we tested the program with multiple books. This increase in input did not lead to an increase in resource usage.

The results of this experiment suggests that the impact of a small helper program is negligible. As such, we choose to make use of this approach that uses the UIAutomation framework. Of course, an experiment of this limited scale cannot be generalised without proviso. To validate this approach, experiments need to be done to measure the impact of such a helper program on a representative scale. These experiments are outside of the scope of this research.

4.4. Communication between host and VM

To make use of the helper program, the host outside of the VM needs to exchange files with the helper program running on the VM. There are various ways for host and VM to exchange files.

In general, files are exchanged via a shared folder. This general method is not suitable for us to use. A shared folder between host and a VM is not a typical shared folder, but a separate channel to pass files. The helper program is therefor not able to see or react to these changes.[3]

Another method to exchange files is to use a shared USB drive. A problem with this method is that the host and the VM cannot simultaneously connect to the USB drive. Extra software is necessary for handling the synchronisation and connections. Although this method could work in a local setup, it does not for a distributed setup, where one host provides input for multiple VMs. Such a case allows for parallelisation of testing. However, this cannot be done on one machine, as the workload of one VM affects the resources available to an other one. A solution for this is to share a USB drive over an IP connection. To accommodate this, extra external software needs to be installed.

A shared USB drive over IP in between the host and VM is not necessary as we can connect

[3] https://forums.virtualbox.org/viewtopic.php?f=2&t=100160

the host and VM directly using a TCP/IP connection. Implementing a TCP/IP connection requires no extra software, but a small addition to the existing host and client program.

As mentioned in section 5.1, we have chosen to implement a simple communication protocol. When the client is ready, it sends a request to the host. The host responds with the next task to be executed until the repository is simulated. When the clients receives the task to replay a commit, extra information is necessary. The textual contents of the commit to be parsed are then requested from the host. Occasionally, a file is encountered that cannot be replay on an IDE (e.g. a JPEG). The clients then makes a request for the file and places it in the repository. The corresponding sequence diagram can be found in the appendix B.

Because our program is a proof of concept, no security measures such as encryption or authorization are taken in regards to the TCP connection.

4.5. DEALING WITH THE IDE

The main function of an IDE is to support the programmer with trivial tasks. To do this, an IDE needs to have some knowledge about the programming environment and to work properly, adds rules and restrictions to the environment and actions made by the programmer.

Git as a version control system does not need to have knowledge about the programming concepts, it just saves a file directory and is "dumb" in comparison to an IDE. This difference in knowledge creates additional challenges in translating a Git repository into actions on an IDE. To overcome these challenges, some information extracted from a Git repository needs to be supplemented, deleted or altered to be executed on an IDE. Some of the encountered challenges and solutions are outlined below.

We have chosen to use Eclipse as our IDE as we already have experience with this particular IDE. In Eclipse, Java-classes are contained within a package and a package in turn is contained within the source directory of a project. Next to to the source directory, the JRE library is also present in the project folder. In some Git repositories, the project folder or even the package folder is missing. To overcome this challenge, a default project and/or default package is created when it is missing in the original repository. The extra created folders need to be taken into account with the creation of following Java-classes and files.

Another challenge that requires us to keep track of state, is that Eclipse puts restrictions on names of packages and files. Eclipse does not allow, for example, spaces in the names of Java-classes. Git could not care less if a name of a file contains a space. As a solution, we replace the spaces with an underscore. The replacements also need to be taken into account in future alterations. Next to that, the UIAutomation Framework finds the UI elements via its name. Above changes in the input also impact the way we locate the UI elements.

Eclipse also supports the programmer by making suggestions and predictions about methods to use. This function is called Content Assist and is shown in Figure 4.3. With the right key combinations, Eclipse can unintentionally insert the proposed methods, altering the input that leads to coding errors. Our preference is to leave this function turned on and use it, but this requires us to predict on the fly when such a suggestion is made. This is beyond the scope of this research and as a consequence, we turn this function off in our simulations.

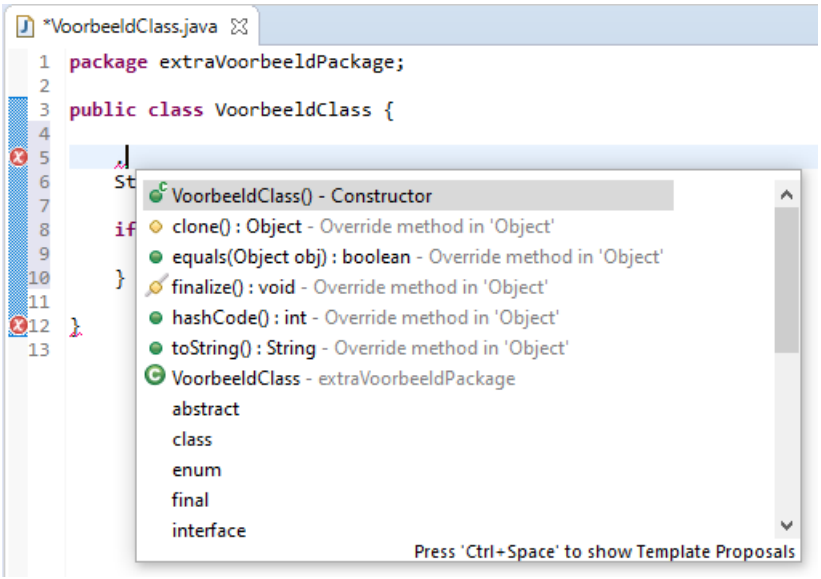


Figure 4.3: Example of suggestions made by Eclipse, a carriage return inserts the suggestion

To help the programmer type, Eclipse provides the programmer with a function to automatically closes and inserts:

- quotation marks
- parentheses
- square and angle brackets
- braces

For example When a brace ("{") is typed and followed by a carriage return, Eclipse inserts the closing brace ("}") automatically on the new line. The closing brace is yet to be encountered by the simulator and also typed when encountered. In the end, a lot of extra symbols are present in the code leading to errors. For the sake of realism, we want to leave this function turned on, but we then have to anticipate when Eclipse automatically inserts the concerning symbol. This is also beyond the scope of this research and this function is turned off in our simulations.

4.6. HANDLING BRANCHING AND MERGING

To accommodate branching in the simulator, we first have to recognize when branching occurs. As previously stated, branching occurs when multiple commits reference the same parent. So we have to check if there is a commit that is referenced as a parent by more than one commit. There are multiple points in the simulation where we can determine if branching occurs.

One approach is to wait until a commit has to be parsed that has the same parent as a previously replayed commit. This approach has some practical disadvantages. It is possible that after a branch has been made, the chronologically next one or more commits belong to the master. When we find out that branching has occurred, the state of the to-be-copied workspace is one or more commits ahead. We thus need to revert to the parent commit where branching occurred before we can begin replaying the concerning commit. While this can be addressed on-the-fly, this adds unnecessary overhead and complexity. Another problem with this approach occurs when juggling multiple branches. In this case, we have to know in which workspace to replay the next commit.

Another approach is to check if a commit is referenced by multiple future commits right after it has been replayed and if so, branch the workspace in advance as needed. This makes sure that all needed workspaces are ready for their next commit and no further actions are required. Via this approach, we can also distinguish workspaces by expected commit so we can always uniquely identify the correct branch.

Once a branch has been discovered, a copy of the project and workspace has to be made. One approach is to literally make a copy of the workspace and switch between the workspaces. But to make branching a practical way of development, the branches need to be merged in to a single project again. Merging two workspaces in Eclipse is practically impossible. As merging is inherently connected to branching, Git provides us with the tools to merge repositories. To use these tools, we have to make repositories out of our workspaces.

Egit is a plugin created for Eclipse that adds a user interface for Git and the Git tool-set to Eclipse, making interaction with a repository from Eclipse possible. We can use this plugin to easily commit the created code, switch to or create another workspace and clone the needed repository to this workspace.

Next to that and with realism in mind, Egit is a very popular plugin recommended for Eclipse used by many programmers. So usage of this plugin adds to the realism of the simulation.

Chapter 5

Validation

The goal of the tool is to automatically replay the development process contained in a Git repository. To validate if the tool does what it is designed to do, we have given it a small repository[1] containing several commits, a branch and a merge to replay.

It took the simulator 45 minutes to successfully complete the entire development history of Git repository. Although a small error was encountered, it did not influence the remainder of the simulation. This result shows that we have achieved the goal of automatically simulating the development process. From this we can conclude that our approach is feasible. The simulation of this repository can be viewed on YouTube[2].

After achieving our original goal, we want to know what the limits of our proof-of-concept are. To test this, we have given it the multiple repositories to replay. We have handpicked repositories from Github that meet the requirements and have chosen to categorize the repositories by number of commits.

- Small: < 10 commits
- Medium: 10 – 100 commits
- Large: > 100 commits

In Table 5.1, we show the results of the repositories simulated. We note the number of lines of code, the number of executed commits and total runtime of the simulations.

[1] https://github.com/jacobbrunson/BasicLighting
[2] https://www.youtube.com/watch?v=YpKmVG1cyBc

Repository	# LOC	# commits	total runtime	Result
Small				
1. ROC-DEV/inleiding-java	11	6	11:24	correctly simulated
2. jacobbrunson/BasicLighting	199	7	42:07	Simulated (with error)
Medium				
3. brianway/java-learning	8	8	16:19	Error while processing
4. EmmanueleVilla/SimpleServiceLocator	1148	8	1:37:48	Error while processing
Large				
5. TheAlgorithms/Java	612	11	1:02:14	Error while processing

Table 5.1: Validation results

Discussion.

2. jacobbrunson/BasicLighting
The repository was simulated successfully, but a small error was encountered. Normally, the simulator takes the .gitignore from the original repository and overwrites the .gitignore that was automatically generated by Eclipse. This did not happen in this simulation as the directory of the repository was not compatible to be simulated. A small adjustment to the directory structure had to be made. As a consequence, the .gitignore did not overwrite the automatically generated .gitignore, but was placed a level lower. The result was that the automatically generated .gitignore was used and the .project file was ignored. This file is necessary to automatically import Java-projects in Eclipse after cloning.

3. brianway/java-learning
In the first couple of commits, no Java-classes were present so no code was written. In the eighth commit, the first Java-class is encountered and the simulator started to code. On the eighth line of the class, a symbol that was not present on the keyboard lay-out had to be typed. The concerning keystrokes send to the application resulted in an unexpected ALT-combination that made changes to the UI. The simulator then could not find the necessary elements and stopped.

4. EmmanueleVilla/SimpleServiceLocator[3] and 5. TheAlgorithms/Java
These repositories both encountered the same problem; unresponsive software. The hardware used could hardly manage running two instances of Windows resulting in a regular occurrence of 100% CPU usage. This made Eclipse run very slow and at times unrespon-

[3]Video of simulation available on request

sive what lead to synchronsation error between the simulator and the UI, stopping the simulations.

Chapter 6

Conclusions and future work

6.1. Conclusions

Usage of a file system changes how data is stored. On a pristine file system, new files are stored consecutively. After a period of use, some files will have grown, others shrunken, old files will be removed, new ones added. All this induces a certain scattering of data over the storage medium. Exactly how data is scattered, is relevant for researchers into file recovery and file system design. Ideally, such data would be taken from systems used in the real world. However, to accurately generalise findings, data on a large number of systems is needed. Moreover, collecting data from a real-world system is labour-intensive. Collecting data from real systems does not scale sufficiently. Consequently, data has been gathered from simulations of computer use. Current simulations typically use a simple model of low-level disk interaction (reading a file, writing a file, deleting a file). The problem with this approach is the fidelity of the model: to what extent does the model capture real-world disk usage?

This research introduced a new method and showed the feasibility of generating sought storage media. The method uses publicly available data on computer use as basis for simulations instead of an underlying model. We have created a method that replays the development of a Git repository. We designed and developed a proof-of-concept that succeeded in replaying different small repositories. The method uses a virtual machine as a system under test, providing an isolated and clean storage medium that can be analysed after the simulation and is easily reverted to an earlier state to replay a simulation under different conditions. The only limitation being a client program installed on the storage medium that is responsible for the interaction with the user interface of the VM and IDE. To mimic a user, the commits are executed on a popular IDE through its user interface. A typing algorithm, developed by others, is used to program with a basic typing rhythm using pause time, keystroke burst, and

a break schedule. The proof-of-concept successfully replayed small (five to ten commits) Java repositories with a proper directory structure on Eclipse as the IDE.

The output of the method is a local copy of the input repository. Analysis of the input repositories and locally generated output repositories showed the feasibility of this approach. This method has the potential to be used in situations where simulators with a mathematical basis are inadequate, such as the previously described creation of storage media patterns. Extra research is required to show how close simulator approximates real fragmentation patterns and to identify other situations where this approach is more applicable than other approaches.

6.2. Future Work

Many implementation and improvements are left to the future due to the amount of possibilities to improve en extend the method.

6.2.1. Research

This research mainly focuses on Java repositories executed on Eclipse. As described in section 4.5, there are differences between Eclipse and Git regarding knowledge about the programming paradigm. Although most of the encountered conflicts have been resolved in the software, we encounter new unexpected erroneous situations with every new repository we simulate. As simulations of large repositories (>100 commits) take hours if not days to simulate, a lot of time is needed to discover, categorize and solve all of the possible erroneous situations.

Next to that, we have only tried to simulate Java repositories on Eclipse. To fully capitalize on the data contained on Github, support for other programming languages and IDEs should be added. Different programming languages and IDEs implement different concepts and structures. This might lead to more or less erroneous situations. Extra research is necessary to discover these concepts and structures and to write new protocols for the simulation of the programming process with new programming languages on different IDEs.

The biggest shortcoming of the simulator is that it does not compile the replayed code as there is not a general method of adding libraries to a project in Java/Eclipse. To overcome this, a different programming language or IDE need to be used that do have a general and centralized method of adding libraries.

As a consequence of the method we use to feed keystrokes to the IDE. We can only feed keystrokes matching the functional layout of the keyboard. For example, we can not feed Korean symbols when the functional layout is set to US International. As many repositories contain for example Asian or Cyrillic characters, more research is necessary to find a

method that does not care about the nature of characters. Thus widening the pool of possible repositories to simulate.

To further widen the pool of possible repositories, support for projects made with Gradle of Maven should be added. As these framework impose extra rules on directory structure and require certain files (e.g. pom.xml in Maven) to exist, extra research is necessary to implement these frameworks.

Interaction with the user interface is done with a client program installed on the storage medium, contaminating the created storage medium. We have already established that context is necessary and that can only be found within the VM. More research has to be done to decrease the impact on the storage medium, while maintaining the same functionalities.

6.2.2. ENGINEERING

As this is the first version of a simulator taking a Git repository as input, many improvements can be made.

The scale of simulations can be extended. With the current simulator, a single repository is simulated at a time. The host of the simulator could be extended with the possibility of controlling multiple VMs, thus generating multiple storage media simultaneously.

We want the simulator to produces user-behaviour workload. To better approximate this type of workload, different functionalities can be added to the simulator:

- When the simulator starts a simulation, it does not stop until the entire repository is simulated. Breaks between programming sessions should be added, such that the simulator stops and/or shuts down for a while after working for some amount of time.
- No programmer starts an IDE, programs for a couple of hours and at the end of the workday, shuts down. Every programmer encounters problems that requires Google or StackOverflow. A function should be added that browses to popular programming website at random intervals.
- Users also listen to music while working. Streaming services are very popular these days and a function should be added to turn on/off a streaming service and change music.
- Typo's do not exist in the current simulator, all the code is copied from the original repository. A function could be added to make typo's while coding and correcting them later on.
- In the current simulator the programming is done linear, but in a regular programming process, programming is done non-linear. Classes, methods and attributes are inserted where they are necessary. A function should be added to make the programming non-linear.

Bibliography

[AAM04] Spyros Antonatos, Kostas G Anagnostakis, and Evangelos P Markatos. Generating realistic workloads for network intrusion detection systems. In *Proceedings of the 4th international workshop on Software and performance*, pages 207–215, 2004. 16

[AS10] Abdolreza Abhari and Mojgan Soraya. Workload generation for youtube. *Multimedia Tools and Applications*, 46(1):91, 2010. 16

[BV14] Sebastian Bauersfeld and Tanja EJ Vos. User interface level testing with testar; what about more sophisticated action specification and selection? In *SATToSE*, pages 60–78, 2014. 16

[BW02] Mudashiru Busari and Carey Williamson. Prowgen: a synthetic workload generation tool for simulation evaluation of web proxy caches. *Computer Networks*, 38(6):779–794, 2002. 16

[CMCU04] Ramkumar Chinchani, Aarthie Muthukrishnan, Madhusudhanan Chandrasekaran, and Shambhu Upadhyaya. Racoon: rapidly generating user command data for anomaly detection from customizable template. In *20th Annual Computer Security Applications Conference*, pages 189–202. IEEE, 2004. 16

[Fis01] Gerhard Fischer. User modeling in human–computer interaction. *User modeling and user-adapted interaction*, 11(1-2):65–68, 2001. 16

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 23–32. IEEE, 2003. 17

[Gan95] Gregory R Ganger. Generating representative synthetic workloads: An unsolved problem. In *in Proceedings of the Computer Measurement Group (CMG) Conference*. Citeseer, 1995. 16

[GDS+03] Krishna P Gummadi, Richard J Dunn, Stefan Saroiu, Steven D Gribble, Henry M Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, 2003. 16

[GVUK06] Ashish Garg, S Vidyaraman, S Upadhyaya, and K Kwiat. Usim: a user behavior simulation framework for training and testing idses in gui based systems. In *Proceedings of the 39th annual Symposium on Simulation*, pages 196–203. IEEE Computer Society, 2006. 16

[HCSJ04] Félix Hernández-Campos, F Donelson Smith, and Kevin Jeffay. Generating realistic tcp workloads. In *Int. CMG conference*, pages 273–284, 2004. 16

[HHK00] Helmut Hlavacs, Ewald Hotop, and Gabriele Kotsis. Workload generation by modeling user behavior. *Proceedings of OPNETWORKS 2000*, 2000. 16

[HK99] Helmut Hlavacs and Gabriele Kotsis. Modeling user behaviour: A layered approach. In *MASCOTS'99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer en Telecommunication Systems*, pages 218–225. IEEE, 1999. 16

[JSL+05] Daniel R Jeske, Behrokh Samadi, Pengyue J Lin, Lan Ye, Sean Cox, Rui Xiao, Ted Younglove, Minh Ly, Douglas Holt, and Ryan Rich. Generation of synthetic data sets for evaluating the accuracy of knowledge discovery systems. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 756–762, 2005. 16

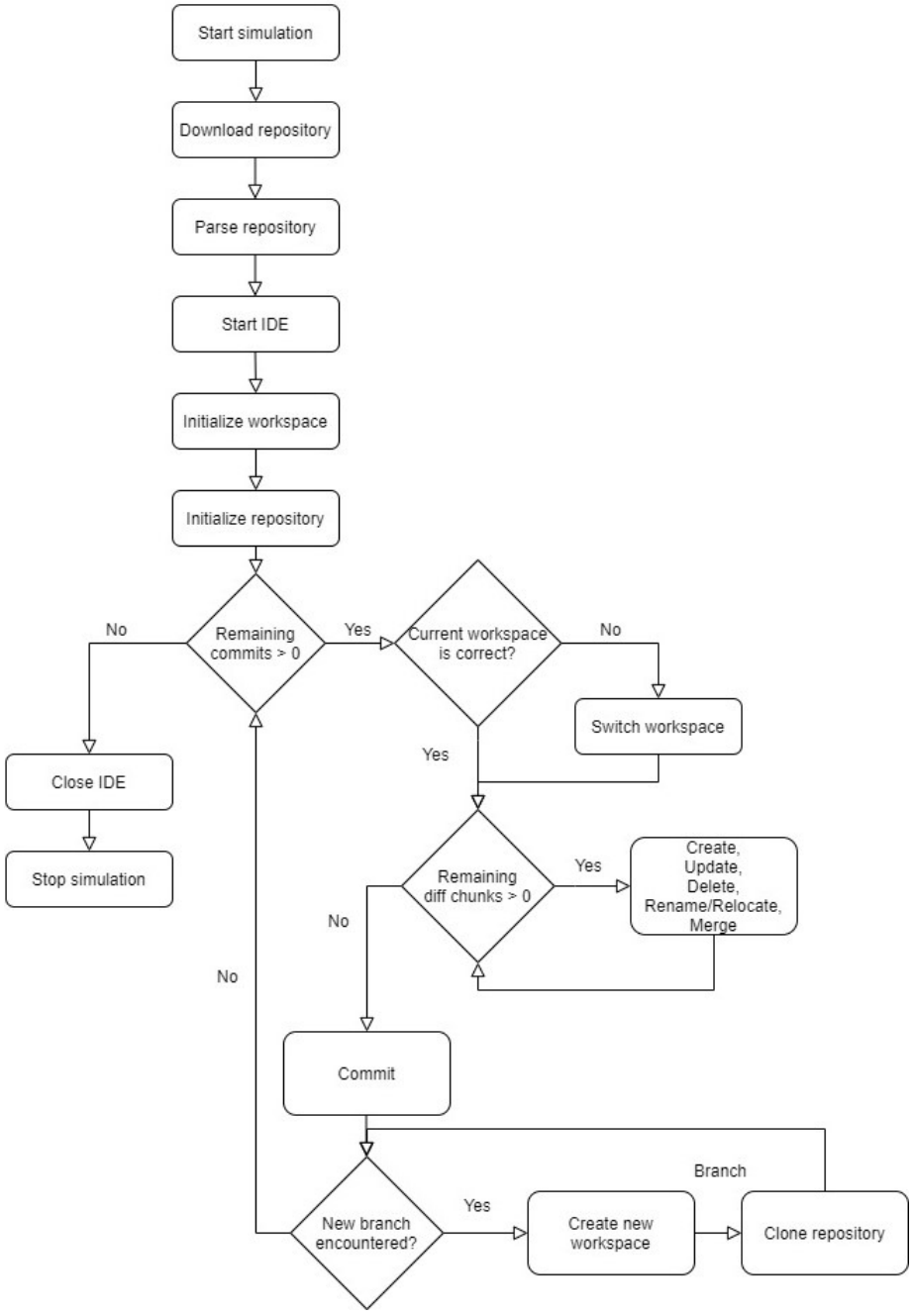
[Noo19] Robbert Noordzij. Synthetic fragmentation experiments using wildfragsim. *Open Universiteit*, 2019. 5, 17, 20

[VAM+02] Eveline Veloso, Virgílio Almeida, Wagner Meira, Azer Bestavros, and Shudong Jin. A hierarchical characterization of a live streaming media workload. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 117–130, 2002. 16

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *MSR*, volume 4, pages 2–6, 2004. 17

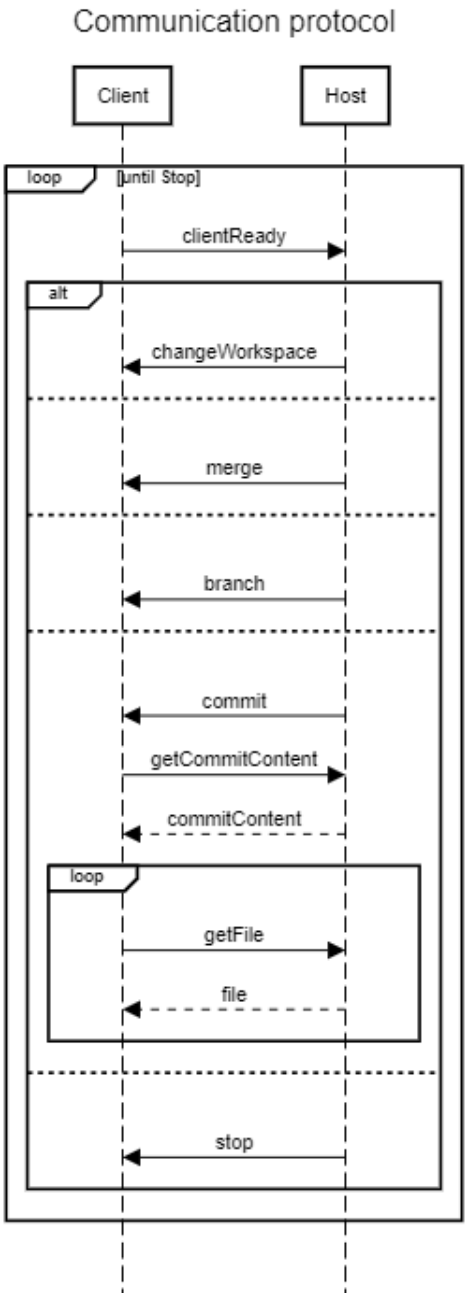
Appendix A

Simulation procedure


Start simulation
Download repository
Parse repository
Start IDE
Initialize workspace
Initialize repository
Remaining commits > 0
No
Yes
Current workspace is correct?
No
Switch workspace
Yes
Close IDE
Stop simulation
Remaining diff chunks > 0
Yes
Create, Update, Delete, Rename/Relocate, Merge
No
No
Commit
New branch encountered?
Yes
Create new workspace
Branch
Clone repository

Appendix B

Communication protocol


Communication protocol
Client
Host
loop
[until Stop]
clientReady
alt
changeWorkspace
merge
branch
commit
getCommitContent
commitContent
loop
getFile
file
stop

Appendix C

Implementation difficulties

C.1. AUTOMATIONID

The AutomationID property uniquely identifies an AutomationElement from its siblings and can be used to find the needed AutomationElement in the tree [1]. In theory, developers of Windows applications give an AutomationIDs to UI elements to make their application compatible with the UIAutomation Framework. In practice, a different reality is true. AutomationElements often lack an AutomationID and with the majority of AutomationElements that do have an ID, the ID appears not to be unique or changes every time the element is created. The use of the AutomationID therefor is not practical and a combination of other properties is necessary to uniquely identify the correct element.

C.2. ONEDRIVE

Onedrive is a service from Windows Live and comes with every Windows installation. It is used to save files in the cloud or to easily share them with others. Files are by default saved in Onedrive and even the desktop resides in Onedrive. As mentioned, the desktop is the root of the tree of the UIAutomation Framework. It appears that, when Onedrive is enabled, the AutomationElements can no longer be found programmatically and no longer matches with the information given by AccessibilityInsights. To be able to execute the simulations, we have disabled Onedrive on the VM.

[1] https://docs.microsoft.com/en-us/dotnet/framework/ui-automation/use-the-automationid-property

C.3. LIMITATION OF USED HARDWARE

In the beginnen of this research, I tried installing an image of Windows 10 on a VM running on my personal machine. After installation, the VM and the containing instance of Windows10 ran very slowly. The cause was an insufficient amount of RAM. My personal machine had 8GB of RAM installed and a single instance of Windows10 required a minumum of 4GB. An increase of RAM was necessary and I increased the amount of RAM to 24GB. Windows now ran a lot smoother. But after installing all the necessary software and starting Eclipse on the VM, I again noticed a drop in performance. It appeared that CPU-usage was at a 100%. This was a problem I could not overcome without buying a new laptop and was the cause for a great amount of erroneous situations. Eclipse regularly did not respond in a timely fashion what caused the simulation to fail.

Appendix D

Reflection on process

In the beginning of the project, the prospect of the amount of work I had to put in, demotivated me massively. As writing was never a favourite thing for me to do, I lacked the needed writing skills and wanted this paper to be finished as soon as possible. I expected from myself that every sentence I put to paper was instantly perfect. My expectations did not line up with reality. I had a lot of frustrating nights, sitting hours in front of the computer not knowing what to do or write. Every night without progress decreased my motivation even more. I had never done such a project before and I did not know where to start or how to do it.

My supervisor encouraged me to stop writing and begin to create a working proof of concept. Throughout the development of the proof of concept, I started to understand the problem more and more. The better I understood the problem, the more I could scope and mold the research. When I had a working proof of concept, I understood the problem and had all the information to start writing. But even with an better understanding of the problem, I had a lot of trouble putting my ideas to paper. I wanted to share a lot of very detailed solutions of problems I encountered throughout the development of the proof of concept. My supervisor made me realize I needed to do a step back and write in terms of concepts instead. Step by step, the paper started to take form. Although the frustrating nights did not disappear entirely, I could handle my frustrations more with the progress I had made.

After the choice to use the UIAutomation Framework, part of the .NET Framework, I chose to develop the proof of concept in C#. I had no experience with C# prior to this research. In the beginning I had a hard time configuring the environment correctly. Many frustrating hours were spend on downloading and handling NuGet packages and assembly references. But through solving the problems I encountered, my understanding of Windows and .Net has greatly increased. So all in all I am very glad that I chose to use a language I hadn't use

before and it made me in general a better programmer.

This research started a few weeks prior to the start of the Covid-19 pandemic, which impacted our lives massively. Working from home, curfew and a lack of social gatherings had an impact on my mental health. The combination of the Covid-19 situation and the frustration I experienced working on the research, resulted in a complete lack of motivation and put progress to a full stop for a couple of months.

All in all, this process required me to make a huge step in writing, which I believe I made. And next to that, it helped me handle my frustrations more and grow as a person. The writing process was a hellish experience in which I had multiple moments where I wanted to quit. I experienced my entire educational career from elementary school up to this point as torture and I would have never thought that I would come this far, thus it makes me so proud that I persevered and finished it.