# Strategy based genetic algorithms approach in automated GUI testing

Stoyanov, Nikolay

*Award date:*
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

# Strategy based genetic algorithms approach in automated GUI testing

*Master Thesis*

Nikolay Stoyanov

Supervisors:
Dr. Yaping Luo (TU/e Supervisor)
Kevin van der Vlist (ING Supervisor)
Dr. Cassio de Campos (Committee Member)

Other supervisors:
Prof. Dr. Tanja E. J. Vos (OU)
Dr. Pekka Aho (OU)

Eindhoven, September 2020

# Abstract

GUI testing is often a human executed and expensive process. Thus, a lot of directions are explored in order to automate it. One of those directions is using completely automated tools for random testing. Unfortunately random testing rarely satisfies the testing criteria. This thesis continues prior work on a genetic algorithms approach which uses a strategy in order to provide "intelligence" as an alternative to the random mechanism. As an addition to the prior work we developed new fitness functions and evaluated them in order to both explore new evaluation criteria and prove the value of the approach.

# Acknowledgements

First of all, I want to thank Yaping Luo for creating all the amazing opportunities for me over the past year and guiding me through my projects as well as my post university path. Without her I would not have been able to do this project which turned out to be quite interesting for me, as well as find the great job following my graduation. I would also like to thank Kevin who would always help and lend me his technical expertise on topics that I do not understand. He is the first person that I met who can always follow my reasoning, which is usually not the case due to my disorganized nature. His knowledge and logical thinking really inspired me regarding the type of engineer that I want to become in the future. I would like to thank Tanja and Pekka who even though are not my official supervisors were always helpful and provided valuable input throughout my project. I also wish to thank Fernando for always providing quick fixes and assistance when I was having trouble with the TESTAR tool and Guy for sharing his work so that I could continue his research. Finally, I want to show my appreciation for my parents who supported me through my five years of university and without who I would not have been able to achieve all that I have.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis is executed under the supervision of ING as part of the Industrial-grade Verification and Validation of Evolving Systems (IVVES) project, which is a part of EU ITEA 3[3]. IVVES is an international project which consists of 32 partners spread over five countries that is running since October 2019 and will finish in September 2022. It aims to develop AI approaches for robust and comprehensive, industrial grade V&V of "embedded AI". It will use machine-learning for control of complex, mission-critical evolving systems and services covering the major industrial domains in Europe.[4]

Graphical User Interface (GUI) is the main communication component between a user and software. Therefore, it is indispensable that a GUI should perform as intended. Generally, users tend to deviate from the exact instructions and expectation of the software developers. This entails that a GUI should work flawlessly in all the possible behaviour that it allows. As the GUI tends to accumulate more and more functions over the time, the tests need to accommodate those changes as well which means that more test sequences are needed. The GUI testing was initially, and still is for many companies, a human executed process where the tester is executing a sequence of actions manually [9]. A semi-automated approach is the scripted GUI testing. An example of a scripted GUI testing is to record a test sequence which can then be executed automatically. However, this still requires that a human has created the initial execution. As it is still human controlled, it keeps relying on the testers to create test sequences. Testers need to anticipate the unpredictable behaviour of users or spend a lot of resources to cover every part of the GUI and maintain those tests. In order to lower the expense of testing, as well as improve the testing process, the automation of tests through scriptless testing has been developed. Scriptless testing is a completely automated testing process in which the tests are both generated and executed automatically. This is done through scriptless testing tools, such as TESTAR.

One of the current issues of sciptless testing is that it usually generates the test sequences randomly. This in turn rarely satisfies the testing criteria, as covering all the GUI test elements through random actions usually requires a lot of execution time. Covering all those elements is required, as the users tend to have unpredictable behaviour while interacting with GUIs. In order to move forward, some kind of intelligence needs to be added to automated testing. With the increasing power of the hardware and maturity of Artificial Intelligence (AI) algorithms, it has become a popular approach for improving automated testing. Last et al. [10] argues that the software quality problems are not too different than the other tasks that AI successfully solves and discusses the applications in software testing. As software testing tends to use a major part of the resources in software development for most companies, as confirmed by ING, it is crucial that these resources are fully utilized to achieve customer satisfaction and requirements fulfilment.

One of the AI approaches in scriptless GUI testing is using Genetic Algorithms (GA) for generating test sequences. The GA is an optimization approach from the machine learning branch of AI. It produces a number of random solutions to a problem, evaluates and modifies them until a satisfactory solution has been produced.

This paper will investigate different approaches for improving an already existing application of

AI in automated testing, namely a GA approach used in the automated GUI testing tool TESTAR. The research in the field usually focuses on finding the best sequences of actions to achieve a certain goal, such as coverage of all the possible states of the GUI. Therefore, it treats the sequences of actions as solutions which are optimized through the GA. What makes the approach used in TESTAR different is that instead of sequences as solutions it introduces the concept of strategy. Strategies are combinations of conditions and actions and will be further described in Section 4.1. The advantage of a strategy over sequence is that a test sequence will only benefit from running it once, whereas a strategy can run as many times as needed and introduce a new test sequence on every run. Also if specific sequences are used, if a change in the GUI occurs it could render those tests obsolete.

The previous research which combines GA, TESTAR and strategies [37] uses an evaluation criteria to determine how good a strategy has performed. It determines that by the number of unique states of the GUI that have been visited. The reason for that is that the more states you visit, the more coverage you have achieved in interacting with the GUI. A unique state is a state of the GUI which is being entered for the first time in the current test sequence. The conducted research had good research fundamentals, however due to time constraints it left a lot of directions which need to be further explored.

As the strategy approach seems to have yielded good results with the initial evaluation criteria [37], it would be valuable to see whether it can be improved by introducing different evaluation criteria and try to achieve different testing goals. One of these goals is bug finding. In this paper, we will work on bug finding, that is, to investigate whether a strategy can be focused around finding a specific bug. We would also like to further research whether the strategy approach is performing better than the random one in TESTAR, and whether using GA to generate strategies is better than generating them randomly. Finally we would also like to find out whether using a GA approach and strategies performs better than using specific testing sequences generated by GA. This raises the following research questions:

**RQ:** Can the strategy-based GA approach improve the effectiveness and performance of the existing automated GUI-based testing tool?

In order to answer this question we would like to research what kind of evaluation criteria we can use. After deciding on the new ones, we would need to see whether the GA strategy approach is effective. This can be measured by checking whether the GA strategy approach will perform better than the random action selection mechanism. This will show us whether the strategy is performing better than the random action selection. We will also need to see whether using GA is useful for improving the strategies. In order to check this we are going to compare selecting a strategy through GA and through a random selection. Thus, we can divide the main research question into three sub-questions:

**RQ1.1:** What kind of evaluation criteria can be used in the strategy-based GA approach?

**RQ1.2:** Will the GA strategy approach be effective while using different evaluation criteria?

**RQ1.3:** Is selecting a strategy through GA better than selecting it through random selection?

The rest of this paper is organized as follows: Chapter 2 will provide background information about Genetic algorithms, TESTAR and the implementation framework ECJ. Chapter 3 will discuss the already existing research in the field. Chapter 4 will explain the modifications made to the testing framework, as well as the motivation for the evaluation criteria. Chapter 5 will discuss the performed experiments in this thesis and Chapter 6 will conclude the thesis and discuss the future work.

# Chapter 2

# Preliminaries

This Section will introduce the background information. We will discuss the testing tool TESTAR, the AI approach Genetic Algorithms (GA), the ECJ framework that implements the GA, and how they are connected to each other in the testing framework.

With the widespread use of the Continuous Integration (CI) process in software development, the time for testing has been significantly limited. Therefore, the results of automated tests are expected to be available almost instantly, even as the complexity of the systems under test continues to grow. Generally, the effort in test automation has been put towards the execution of test cases and not so much their automated construction. Although there are tools for unit test generation, the system level testing proves to be difficult to automate, especially in the cases with a modern GUI due to the increased complexity. This is an important part of test automation, as the GUI is the main connection between the user and the software.

Artificial Intelligence (AI) has been a hot topic for discussion in recent years. With time it has made a large impact on the world around us. The algorithms developed for speech and image recognition, information evaluation, data analyses, self-driving cars and computer games players, such as chess, are all results of the development of AI. Since AI has been proven to be useful in different areas where it can replace human intelligence, naturally by widening the applications it reached the software testing area. There are approaches from different areas of AI applied into the software testing field. This thesis is limited to GA, a subdomain of "AI". The reason for choosing GA is that it has ability to efficiently search for solutions in complex search spaces. GA also solves with optimization problems and due to the fact that we aim to optimize the strategy, GA is a good fit for this research.

## 2.1 TESTAR

As it is not feasible to manually create scripts for all the possible actions of a GUI, the scriptless GUI testing tool TESTAR has been developed [36]. It is an open source tool that automates test case generation by using several action selection mechanisms and test oracles. Oracles are the systems that evaluate whether an error has occurred. The approach of the tool is to start the System Under Test (SUT), gather information about the state in which the system is in and perform an action. The goal of TESTAR is to find failures in the GUI through an action-selection mechanism . It does that by performing different actions and exploring different states of the GUI. The goal of this research is to improve that process through the action selection mechanism.

TESTAR achieves its functionality through the accessibility application programming interface (API) [19] of the underlying operating system for local applications and the Selenium WebDriver for web applications [18]. Through the API TESTAR has access to all the information about the GUI elements, which are known as widgets, on the current screen. This allows the full automation of both the creation of test sequences and their execution. TESTAR features a customizable interface which they named protocol, which provides the set of settings, such as what is the action

selection process or which states should be considered as bugs.

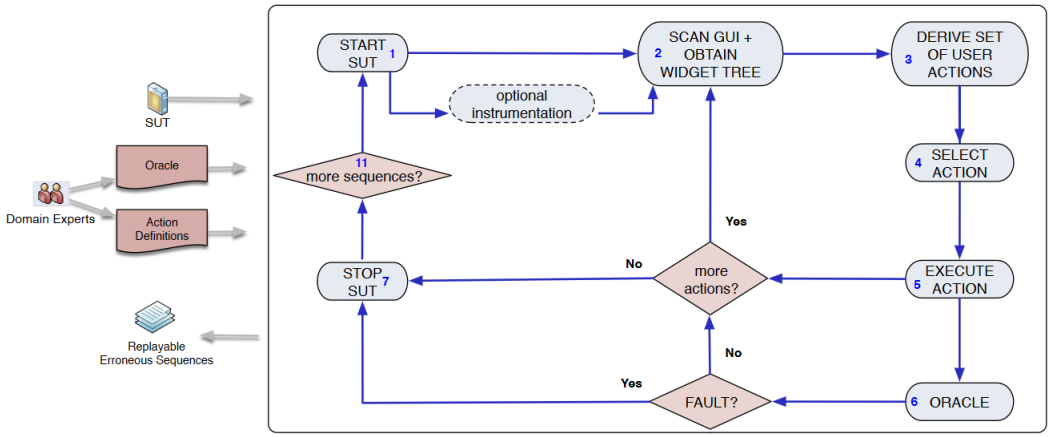


Figure 2.1: The TESTAR approach. Figure taken from [36].

The TESTAR approach is shown in Figure 2.1. A single test execution consists of the following steps:

**Step 1:** Start the SUT.

**Step 2:** Scan the GUI in order to obtain the state of all the GUI elements on the screen, such as type, position, enabled/disabled etc.

**Step 3:** Generate widget tree, the hierarchical representation which allows TESTAR to derive a set of actions.

**Step 4:** Select a promising action based on the protocol setting. In this research, the selection process is done through the notion of strategy which will be explained in Subsection 2.3.1.

**Step 5:** Execute the action.

**Step 6:** Repeat Step 2,3, 4 and 5 until you have reached the maximum number of actions or found a bug.

**Step 7:** Stop the SUT and gather the metrics.

### 2.1.1 Settings

This subsection will explain the general settings that are available through the TESTAR GUI [36].

Figure 2.2 shows the window where you select the general TESTAR settings. The first box is where you specify the SUT. In this example the notepad SUT is chosen to be started through the command prompt. In the second box we can select the sequence duration. The third box allows the selection of a specific protocol. Finally in the fouth box, the application name and version can be given. This allows TESTAR to generate separate state models. By choosing a different name or version, a new instance of state model is generated from the scratch.

The TESTAR filters tab (Figure 2.3) allows the user to prevent specific actions from being executed. For example by filtering "close" we will not allow TESTAR to use the close button, which is present on almost any GUI in Windows. For example we would not want the GUI to close and the sequence to end early, therefore, one would always want to filter out that button.

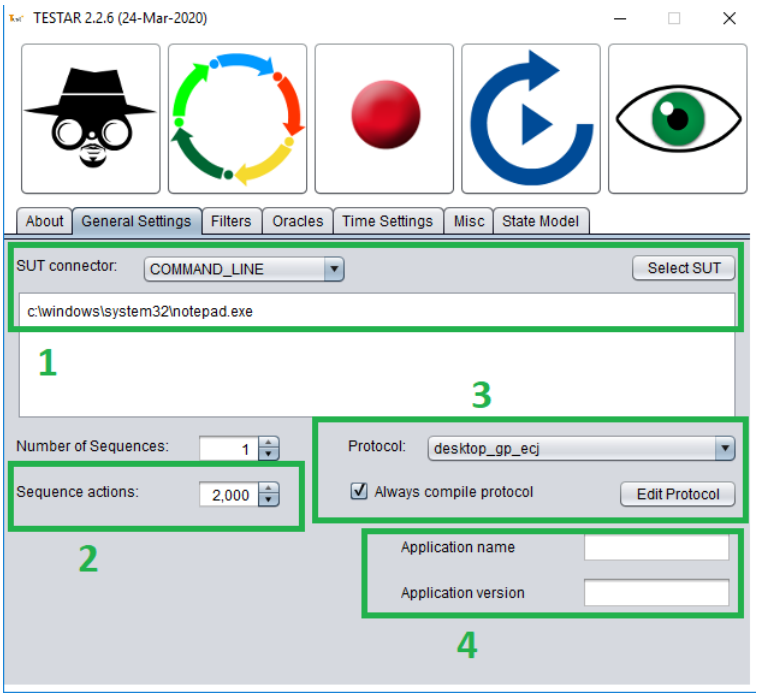


Figure 2.2: TESTAR general settings.

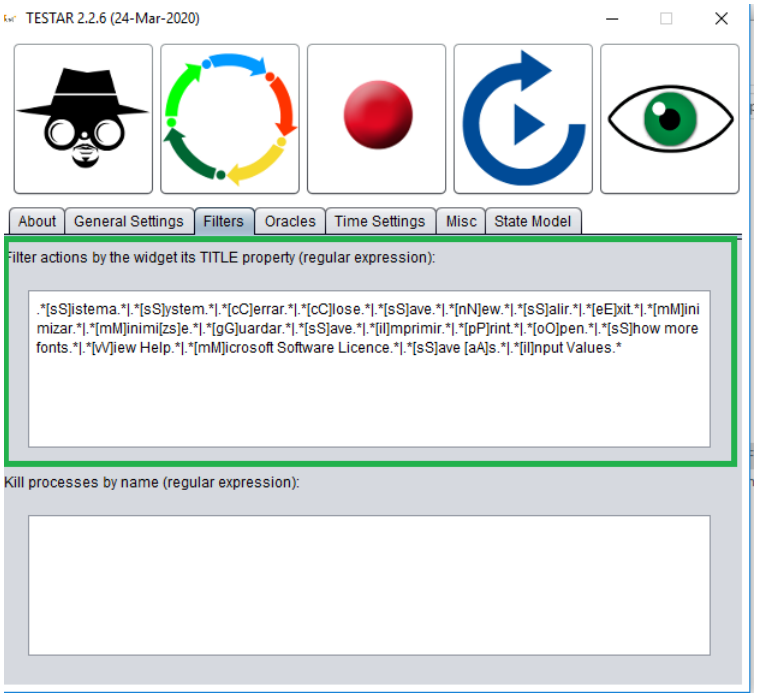


Figure 2.3: TESTAR filters.

In Figure 2.4 we can set the oracle settings for TESTAR. We can see that if a window with the name “error” is encountered, we would assume that a bug occurred.

The time settings tab in Figure 2.5 allows us to set the specific time between actions, as well as their duration. This is required, as sometimes if the actions are performed too fast they are

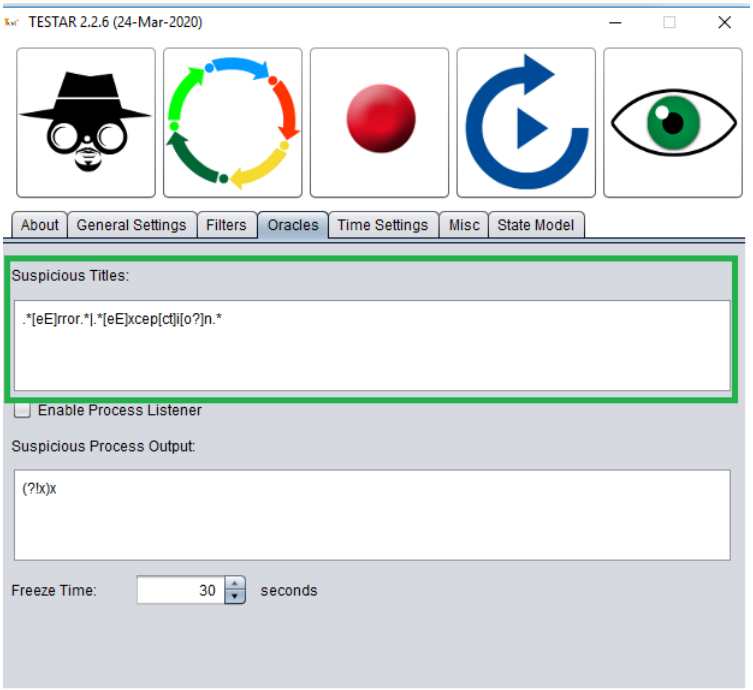


Figure 2.4: TESTAR oracles settings.

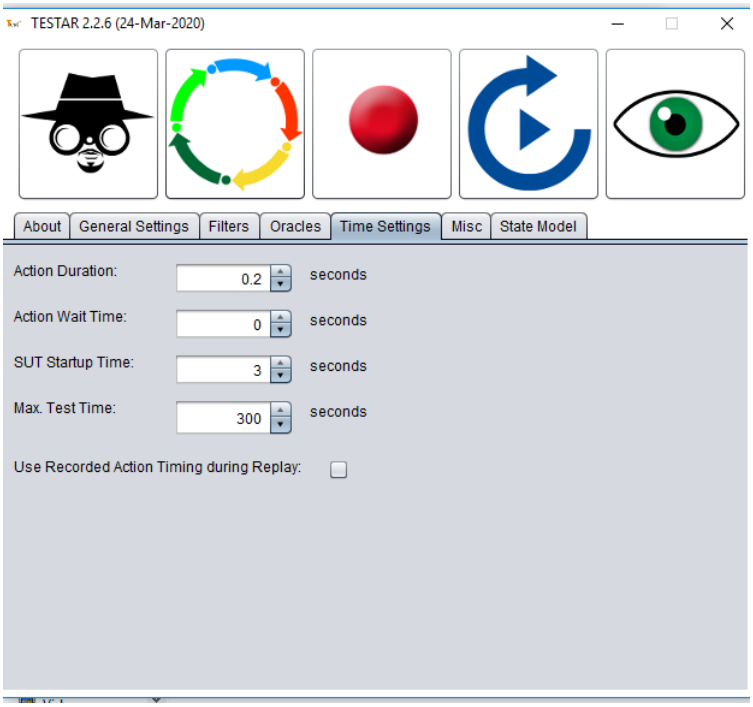


Figure 2.5: TESTAR time settings tab.

not recognized by TESTAR. The SUT start-up time is also important, as some software tends to start slower than others. If not high enough value is set, the software may fail to start on every run and no actions will be performed. The maximum test time allows us to set a time limit on the execution of each sequence. This is useful when one wants to perform a time-limit test. It also

helps in the current experiment as TESTAR will sometimes stop performing any actions.

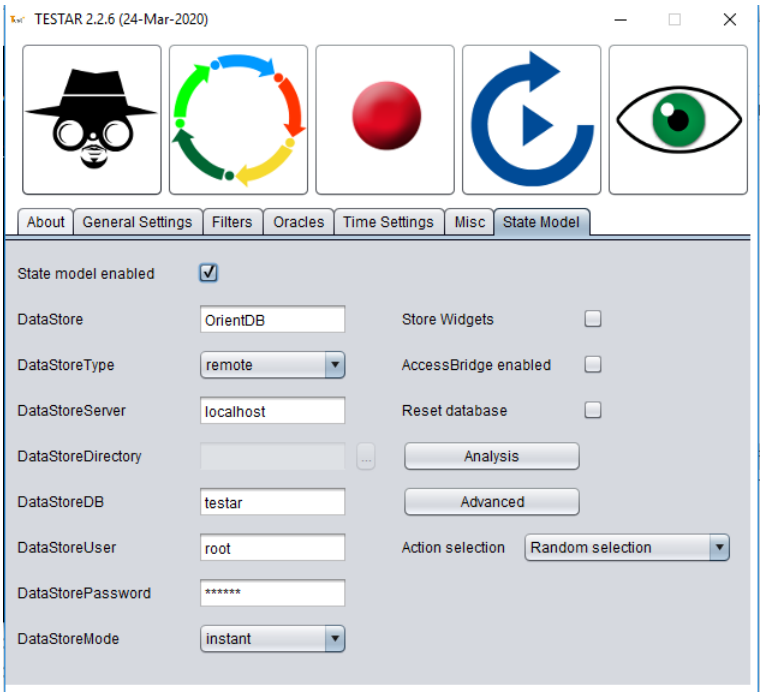


Figure 2.6: TESTAR state model tab.

The settings in Figure 2.6 allows us to connect to a specific database, as well as reset it when needed. It also allows us to analyze the state-model currently generated in that database.

### 2.1.2 State and state model

A state of the GUI is identified by TESTAR through a hash function of the available attributes of every widget on the screen, which are part of the GUI. In case of an unique state only the role attribute of each widget is used [31]. The state model of TESTAR has been built on top of the graph database OrientDB. It is a multi-model open source NoSQL database management system which supports data models in documents, graphs, key/value, and objects [15].

The state model of TESTAR is a more complex version of the event graph model, which is further described in Section 3. The state model is a set of nodes and edges, where the nodes are a set of states of the GUI and the edges are the transitions between those states. As TESTAR gathers the information about the GUI through the API for desktop applications, and Selenium WebDriver for web apps, the state model identifies the state based on two types of information:

- The parameters for each available widget on the screen that the API provides.
- The previous action that caused the system to reach this state.

As the state model of TESTAR provides a lot of information about the states through the attributes of the widgets, different abstraction layers are present. In this thesis we will use the highest abstraction layer as it is closest to the event flow graph model, which is used by most of the researches in the area of GA-based GUI testing. An example is given in Figure 2.7 and 2.8 about how the previous action affects the state model. The example has been taken from Mulders' thesis [29] who developed the state model. The example is based on installing a simple application. You start from the "Desktop" and launch the application to get to the "Welcome screen". There you can cancel the installation which would ask us to confirm or cancel the cancellation leading respectively back to the "Welcome screen" or the "Desktop". From the "Welcome screen" you

can move to the "Licence agreement" screen if you decide to continue with the installation, which again gives us the option to either Agree to the terms or cancel the installation. If you agree to the continue with the installation you move to a "Select installation parameters" screen which again allows you to continue or cancel the installation. Then you move to a "Finished" screen where you can close the installer and move to "Desktop". On Figure 2.7 we can see the state model which only takes into account the current state and on Figure 2.8 where it also takes into account the previous action.

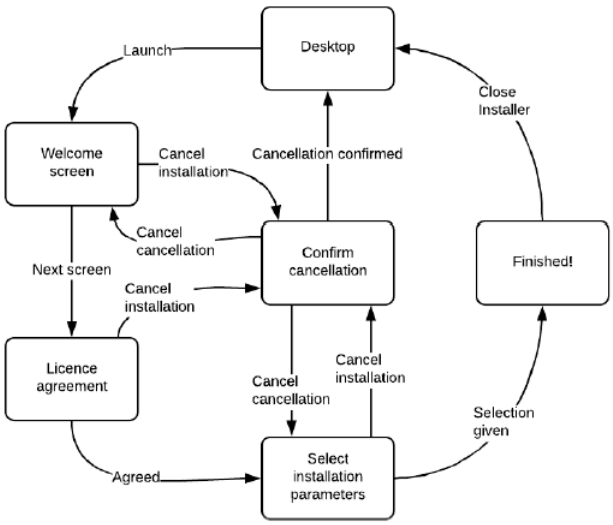


Figure 2.7: State model based only on widget information[29].

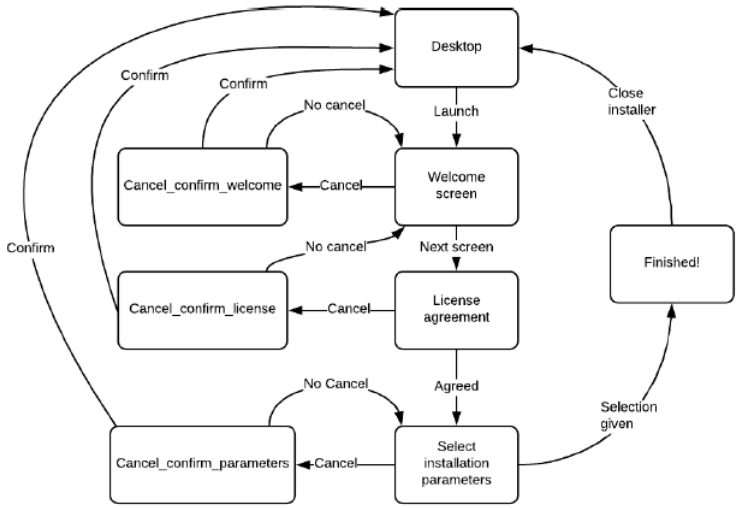


Figure 2.8: State model using predecessor state context information[29].

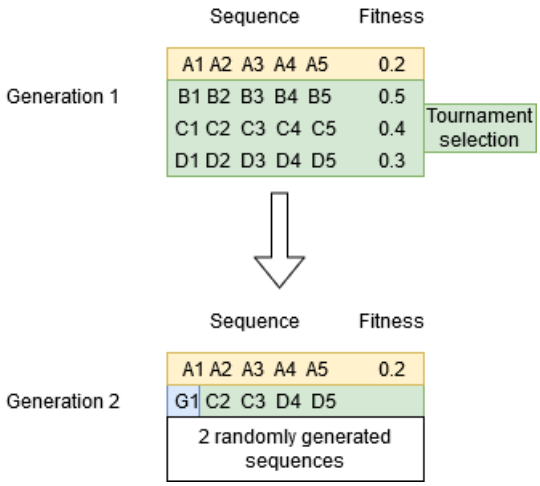


Figure 2.9: Genetic algorithm example.

## 2.2 Genetic algorithm

The Genetic Algorithm (GA) is an optimization approach for generating solutions to a problem. It mimics the process of natural evolution to "evolve" solutions aiming to improve them towards the optimal solution for the given problem. We are dealing with an optimization problem in this thesis, namely performing the best selection of a new action. Therefore, the GA approach should be well suited for this project[28]. Below we will give a short description of the general terms of GA:

**Individual** A solution to the problem that the GA is solving.

**Gene** A feature of an individual.

**Population** In a GA, each iteration, or generation, results in a set of possible solutions and the population refers to the complete set these generated solutions after a given iteration.

**Generation** The current iteration of the GA. Usually population is used to refer to the initial set of solutions, while generations refer to the ones resulting from the following generations.

**Fitness function** The fitness function is the evaluation criteria of the GA. Based on how well a solution has performed it is given a fitness value.

The general approach for GA is to begin with a random set of solutions. The set usually is referred to as a population and the solutions are individuals or chromosomes. Each of these individuals is then evaluated through a fitness function which aims to bias them towards the optimal solution. The best individuals are selected through the a cross-over and mutation processes, which combine and change the features of new solutions are generated. These processes will be explained in the following example. In this way a new generation is produced. The process then repeats producing new generations until either the specified number of generations is reached or an optimal solution has been generated.

An example of creating a new generation through GA is shown in Figure 2.9. The example features a population of four individuals which is the first generation. Most of the researches described in Chapter 3 consider the individuals to be sequences of actions. For the purpose of this example we will do the same. Each of these sequences have already been evaluated and are aiming to minimize the value of the fitness function. Also the approaches Elitism [11] and Tournament Selection [27] will be used in both the example and later in this research. The parameters for the GA example are given below:

**Number of actions per run** 5 as we can see that each solution is the result of five genes.

**Population size** 4 as we have a initial population of four solutions

**Generations** In this example we are only showing the first and second generation.

**Elitism** 1 as we are selecting one individual to be carried out to the next generation without changes.

**Tournament selection** 75% of the population per generation in the tournament to determine which solutions will be used in the next generation. In this case we are creating one offspring from the two best solutions in the tournament.

**Mutation rate** In this case we are mutating one of the five genes, so we can assume a mutation chance of 20%. However usually in GA this is set to around 5% and it will be the value that will be used in this research.

**Elitism Selection** when a number of the best performing individuals is exactly copied in the next generation. It will not be mutated or changed in any way. In this case this is the sequence of actions "A". Therefore the first sequence is directly copied in the new generation, as it has the best result from the fitness function which applies to all. It has been evaluated already, so to save computational time, it keeps the existing fitness value.

**Tournament Selection** when a percentage of all the individuals of the generation are selected, those individuals are compared against each other's fitness values. The best of them are chosen for the process of cross-over. In this example a tournament size of three, or 75% has been chosen. The randomly selected individuals are the last three and the sequences "C" and "D" are selected for a cross-over. We can see that the second sequence of the second generation is combining both the actions from "C" and "D".

The mutation process also occurs in our example. It can not affect the 'elite' sequence but it can affect the second sequence. Even though it is a result of sequences "D" and "C", the first action is not present in any of them, as it is the result of the mutation. It has been coloured in blue ("G1").

After the cross-over and mutation processes are finished, two more sequences are randomly generated, in order to reach the population size of four. Then the three new sequences are evaluated through the fitness function and we have completed our second generation.

### 2.2.1 Java Evolutionary Computation Toolkit

Java evolutionary computation toolkit, also known as ECJ is an open-source genetic programming tool, which is popular in the genetic programming community. To achieve the implementation of GA in TESTAR, the tool is running separate instances of TESTAR test sequences and using the evaluation metrics in order to perform the genetic algorithm on the individuals (strategies) used for those sequences.

## 2.3 Testing framework

This Section will introduce the testing framework that was used in the previous research [37]. It will explain how the protocol deviates from the default one and how the TESTAR and ECJ framework interact with each other.

### 2.3.1 Protocol

A protocol in TESTAR is a Java class that is responsible for executing the different parts of a test sequence loop (Figure 2.1). A protocol can be used to modify the behaviour of TESTAR in specific cases. For example TESTAR can be modified to always close a specific pop-up through the protocol. The default protocol of TESTAR has been extended in order to support a number of functions:

- A pre-defined word list. By default when TESTAR needs to insert a text, random text is being sent to the SUT. This random text is not limited which can cause the text action to take a unnecessary long time [17]. To mitigate this issue, a pre-defined word list has been added to the protocol.
- The protocol has been extended to support the full set of UTF-32 characters.
- A number of metrics are being saved for each run so that a strategy can be evaluated. The metrics are explained further later in this section.
- Instead of following a random action selection, the protocol follows a strategy for action selection, which is explained in Section 2.3.1.

#### Metrics

The testing framework saves a number of different metrics for each sequence that has been executed. They are presented below:

**Unique states** As already discussed earlier, the unique states are determined by the widget information on the current screen. They are also the metric used for the calculation of the fitness function in the research by Theuws [37].

**Unique actions** Similarly to the unique states the system saves the number of unique actions that have been executed.

**Unique states per action** Another metric being saved is the number of unique states that have been introduced up to a specific actions. For example in the same sequence. Suppose in the first 10 actions and 4 unique states have been identified, then a new action can be executed which results a new unique state. The metric will indicate that by saving that a new state was introduced on action 11.

**Bug severity** Through the severity information we can see if a bug has been found in the execution.

#### Strategy

As it has already been mentioned both the previous and current research focuses on exploring different strategies as an alternative to evaluating specific sequences. A strategy is an action selection mechanism which is a combination of "if-then-else" statements. The strategy can be also presented in a tree format, which will make it easier to show how the GA processes are applied to it. An example tree is shown in Figure 2.10. The strategy depicts the following:

- If condition X holds, then execute action X.
- If condition X does not hold then if condition Y holds, then execute action Y.
- If condition Y and X do not hold, then execute random action.

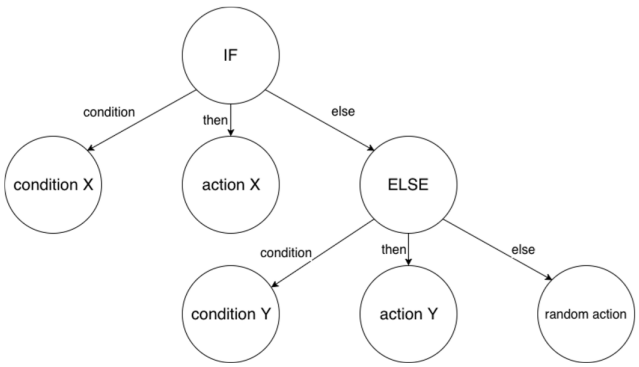


Figure 2.10: An example of a strategy tree in TESTAR.

The strategy will always return an action based on the widget information of the GUI in the current state. In the implementation of the strategy tree in the ECJ framework the Extended Backus-Naur Form (EBNF) has been used. EBNF is a metalanguage which in this case helps to easily translate the strategies to a java format[14]. The grammar of an example is given in Tables 2.1 and 2.2 if one wants to get more information about each type of node in the strategy tree. On the example you can see that there are 2 different strategies types which can be generated, a simple statement (MG) one or a more specific one (EARV strategy). Those two types were derived from the previous research in the field [7], [12]. Underneath each data type, for example action, you can see the specific types of actions that are the values it can take, such as a random-action. The tree strategy shown in Figure 2.10 is consisting of an EARV strategy. The conditions are of the type *paren_expr* which in turn is of the type *boolean* and the actions are of the type *expr* which is in turn of the type *action*.

```
MG strategy
: statement
;
EARV strategy
: ' if '  paren_expr  ' then '  expr  ' else '  expr
;
statement
: ' if '  paren_expr  ' then '  statement  ' else '  statement
| expr ' ; '
;
paren_expr
: ' ( '  boolean  ' ) '
;
expr
 : action
 ;
boolean
 : number  ' greater-than '  number
 | number  ' equals '  number
 | boolean  ' and '  boolean
 | boolean  ' or '  boolean
 | ' not '  boolean
 | ' type-actions_available '
 | actionType  ' equals-type '  actionType  †
 | ' left-clicks-available '  †
 | ' drag-actions-available '  †
 | ' state-has-not-changed '  †
 ;
```

Table 2.1: EBNF strategy example from [37].

```
action
 : ' random-action '
 | ' previous-action '
 | ' random-action-of-type '  actionType
 | ' random-unexecuted-action '
 | ' random-unexecuted-action-of-type '  actionType  †
 | ' random-action-of-type-other-than '  actionType  †
 | ' random-least-executed-action '  †
 | ' random-most-executed-action '  †
 ;
actionType
 : ' click-action '
 | ' type-action '
 | ' drag-action '  †
 | ' hit-key-action '  †
 | ' type-of-action-of '  action  †
 ;
number
 : ' number-of-action '
 | ' number-of-left-click '
 | ' number-of-type-actions '
 | ' number-of-drag-actions '  †
 | ' number-of-previous-executed-actions '  †
 | ' number-of-unexecuted-type-actions '  †
 | ' number-of-unexecuted-left-clicks '  †
 | ' number-of-unexecuted-drag-actions '  †
 | ' number-of-action-of-type '  actionType  †
 ;
```

Table 2.2: EBNF strategy example from [37] continued.

### 2.3.2 Research set-up

The generic architecture of the framework for testing can be seen in Figure 2.11. The architecture generates a list of strategies in the ECJ framework by using genetic algorithms. These strategies are then send to a TESTAR instance for testing, which uses them to test a SUT. After finishing the testing the list of metrics is saved and send back to ECJ in order to evaluate the strategies. After evaluating the list of strategies the ECJ framework generates a new generation (list) of strategies and the process is repeated until the maximum number of generations is reached.

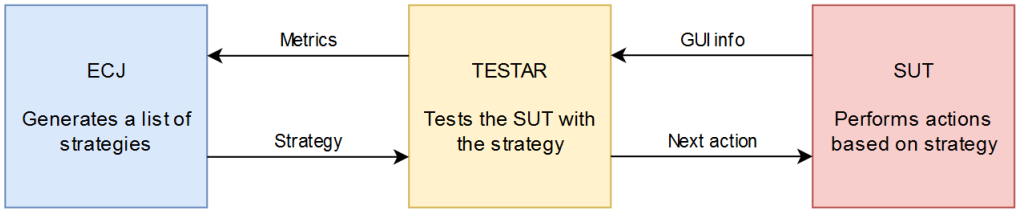


Figure 2.11: Overview of the generic architecture of the testing framework developed by Theuws [37].

The framework was uploaded to Github [2]. Replicating the framework was a hard and lengthy process due to various system details and complications, but with the help of the author and the TESTAR team we were able to do so.

# Chapter 3

# Related work

In this Section we will describe the related work in GA based GUI testing.

## Advantages of automated GUI testing

Last et al. [10] already argued that AI can be applied to software testing, as the problems are not too different than the problems that AI generally tends to solve. The authors provided a good summary of numerous applications of AI in software testing. Some of the researches include developing test oracles cause-effect graphs and fuzzy logic, predicting faulty modules or overcoming the problem that test cases are inexcusable during regression testing. One of the topics that their research does not cover is test sequences generation which is going to be the focus of our research.

The GUI is important part of the software-user interactions, as it is the connecting factor between them. Therefore it is crucial to prevent failures in the functionality of the GUI. Software testing also usually consumes a major part of the software development resources. According to Alégroth et al. [9], these costs consume up to 50% of the software development costs, where 20% of the total costs are directed towards manual GUI testing. The authors performed a research in two companies: Siemens and Saab. According to the research both companies developed a semi-automated scripted testing approach. Both companies reported that they still retained 60% of the original GUI testing resources in maintaining the scripted testing approach. Although over time the Return Of Investment (ROI), depending on a number of different factors, proved to be beneficial for the companies, it was argued that the approach is not feasible as the trust in the quality of the semi-automated tests is lower than that of the human executed ones.

A step forward in automated GUI testing will be fully automating the test cases by automating both the generation and execution of test cases. This will provide drastic decrease of the software development cost and increase of the ROI. By improving the test case generation with AI the user satisfaction and trust in quality will also improve if reasonably good software test cases are resulting from the automation.

## Previous work in the GA strategy approach

The current research will extend an already conducted research [37] by evaluating on different fitness functions. Key-words in the research and what distinguishes it from the other research in the field is the strategy and genetic algorithms approach. The goal of the this research is to explore whether the approach is suitable for different fitness functions and how does it compare to the already existing GA approaches in GUI testing. The first research conducted by Vos et al. [7] was using a simplified version of the strategy by limiting the tree depth. A follow-up research performed by Groot [12] was aiming to extend the strategy by allowing a larger variance and size of the strategy. An unexpected result was that the approach did not outperform the approach from the simpler strategy. It also failed to outperform the best random strategy that it found in the same number of generations. However, due to different SUT and artifact versions, it is difficult

to compare these researches and draw strong conclusions. Theuw's research was aiming to re-evaluate the research [12] by Groot by having more SUTs in common with the previous research. The results conclude that the approach with an extended strategy significantly outperforms the simpler strategy. However, a downside is that selecting a best random strategy from a set of populations generally outperforms the GA approach. However, the author argues that by having a longer evaluation the GA will improve with more generations. This thesis will mostly focus on exploring different fitness functions for evaluating paths in GUI testing. Sometimes a fitness function will be explained as a reward system. This means that if a certain action is rewarded, the fitness function produces favourable results if that action is included.

All of the previously conducted experiments used the same evaluation technique. They were aiming to explore as many unique states of the SUT as possible. Due to the promising results of the extended strategy, it is worth to explore whether different goals can be achieved through it. To achieve this the current research focuses on exploring different fitness functions for evaluating GA-based GUI testing.

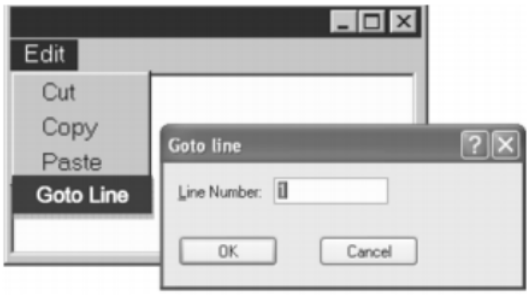


Figure 3.1: Example GUI taken from[23].

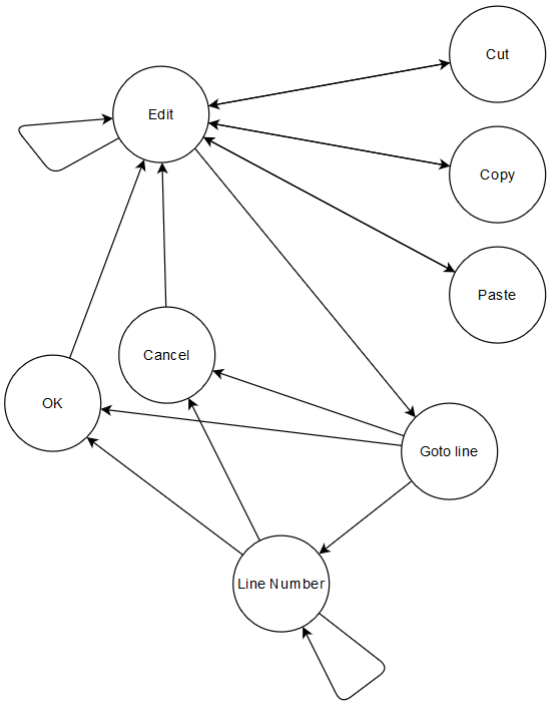


Figure 3.2: Example event-flow graph for Figure 3.1.

## GA based GUI testing

Using the Event-Flow Graph (EFG) model [34] [20] is the most popular approaches in GA-based testing. An EFG is a graph where the nodes are a GUI events and the edges are directed towards other GUI events available once the GUI event has occurred. An example of an event-flow graph of the simple GUI inf Figure 3.1 can be seen in Figure 3.2. These approaches usually produce specific sequences of actions as individuals in order to achieve high coverage of the event flow graph. An example is given by Rauf and Anwar [30] which generates an event flow graph model and bases the fitness function on the number of explored paths (edges of the graph) divided by the length of the sequences. They give coverage analysis on notepad. Although our research's goal is to produce a strategy, which is then used to produces sequences, a comparison between our approach and the one of Rauf and Anwar will help in evaluating whether a strategy outperforms a specific sequence. This experiment is described in Subsection 5.2.

Similarly to the previous approach, Srivastava and Kim [35] argued that it is not possible to achieve a high test coverage and adopted a different approach. Instead of focusing on improving the coverage of the graph, their research focused on exploring the notion of "critical paths" in the graph. The critical paths were defined as loops, branches etc. Although the results of our research and the ones in the paper are not comparable due to the state model of TESTAR, the research provides an example on how automated testing can be tuned to address different goals of the testing process.

Perhaps one could argue that exploring all the paths to achieve high coverage is redundant. Consequently, due to the redundancy, automated testing also becomes a time-consuming task. Ghiduk et al. [16] developed an approach to avoid repetition. The study introduced the notion of dominant paths. A dominant path is a path in the event flow graph model which follows a straightforward path. It does not allow loops or going through the same nodes through different edges. These dominant paths also aim to simulate existing test cases, therefore focusing on the actual requirements and happy flow of the program. The research was directed at rewarding the sequences which follow these paths, which in turns reduces the repetition of states and makes it more time efficient. However, going to a certain state with a different previous state could introduce new bugs. In order to tackle this problem, we would like to increase randomness, so a transition between states with different starting states is rewarded in our research.

Laţiu et al. used event-flow graph and GA by evaluating the paths based on the number of GUI changes and gave a small penalty if you enter the same state more than once [24]. This could be considered similar to evaluating the number of unique states that you visit. However, if you evaluate based on the number of unique states you encounter, you would not be rewarded if you enter the same state twice, while the research by Laţiu et al. gave a diminished reward. Another research was performed by the same authors with similar setting in different GUI applications about water monitoring [22]. Although both researches note improvement of the fitness function with generations, they do not provide a comparison about how good their approach is compared to random testing or other evaluation techniques.

## Other approaches with relevant fitness functions

Ahmed et al. [8] proposed an approach different than the rest of the optimization problems in GUI testing by using combinatorics. Although the research used a swarm particle optimization algorithm similar results can be acquired by using GA. However, the difference from the other researches is that instead of evaluating the fitness functions through some coverage based on the GUI, the authors tried to achieve all the possible sequence combinations. This resulted in a high coverage as the different combinations provided new sets of unique sequences. The approach proposed by Ahmed et al. helpes in developing the idea on how we can focus on faster introduced sequences in our approach described in Section 4.3.

Another research adopted the Ant Colony Optimization(ACO) algorithm in two separate researches [21], [25]. ACO is an optimization algorithm which focuses on doing so by exploring graphs. The behaviour of the algorithm is inspired by real ants. The algorithm "ants" are explor-

ing the possible paths in the graph and evaluating them. As the approach still used an optimization algorithm which evaluates and finds the best path, the fitness function could be applicable to the GA approach. Their fitness function focused on picking the next node in an event-flow graph which has the most outgoing edges. The fitness function is applicable for GA testing in a fully generated event-flow graph.

Prabhu and Malmurugan proposed an event-flow graph approach to address faults[13]. The algorithm is Bee Colony Optimization (BCO) algorithm, which works similarly to the ACO algorithm. The difference is that BCO actually focuses on exploring more rewarding paths, whereas the ACO provides information and decides whether to explore a path based on that information [32]. The research focused on exploring states that introduce different data output than expected and cover as many of these states as possible in a single transition. Although the results look promising, the information about software used is not given. Therefore, we can not draw strong conclusion from their research about the feasibility of finding bugs through GA.

## Summary

Overall the results in the field of software testing by using GA and similar optimization algorithms are promising. They often have different goals and core ideas due to the authors' different conclusions. Some can be compared to the results of this thesis due to similar software and fitness functions but others are too different for comparison. However, they all propose different ideas in evaluating already effective solutions which is the main goal of this research. Therefore, three different functions have been developed. They will be explained in Section 4.3. The fitness function about state transitions in the state graph is similar to the one described in one of the aforementioned researches [35]. The fitness function about curve steepness is inspired by the approach proposed in [8].

# Chapter 4

# Methodology

The goal of this thesis is to perform further research on the approach proposed by Theuws [37]. As the previous work used a single fitness function in this thesis we will try to adopt new ones in order to see whether GA-based strategy approach can be used to further achieve different goals. We would also like the compare the strategy approach to the sequence-based approach proposed by most of the GA researches. The strategy is an action selection mechanism based on the available actions. It is further described in Section 4.1. The architecture of the extended framework is described in Section 4.2. The contribution of this research is making the framework more stable and developing three new fitness functions for evaluating the generated strategies described in Section 4.3 and their related experiments described in Chapter 5.

## 4.1 Strategy vs Sequence

One of the challenges that automated GUI testing is facing is action selection. As mentioned before, sequence generation seems to be a working approach in GA for selecting the next effective action to undertake. However, developing a sequence is usually a time-consuming task that provides you with only one test sequence. This means that a genetic programming approach will result in a small number of test sequences. In order to generate the sequences on the fly and provide more possible effective sequences with genetic programming, the notion of strategy is introduced. A strategy is a combination of if-then-else statements which always return an action to execute in any state of the GUI. It has been developed in a way that it can be represented in a tree, so that the GA approaches can be applied to the nodes in the tree. An example of a tree can be seen on the Figure 4.1. The maximum tree depth in this thesis is going to be 17 as it was the value chosen in the previous research.

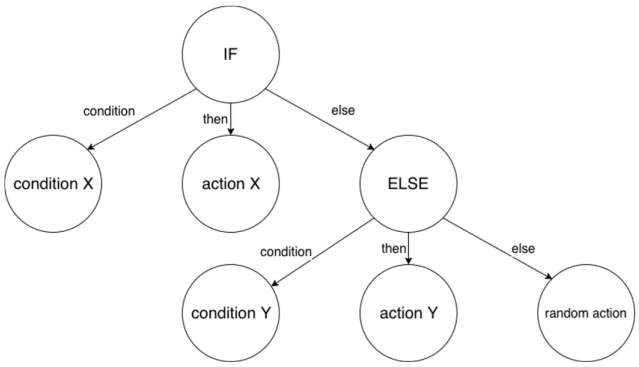


Figure 4.1: Strategy Tree

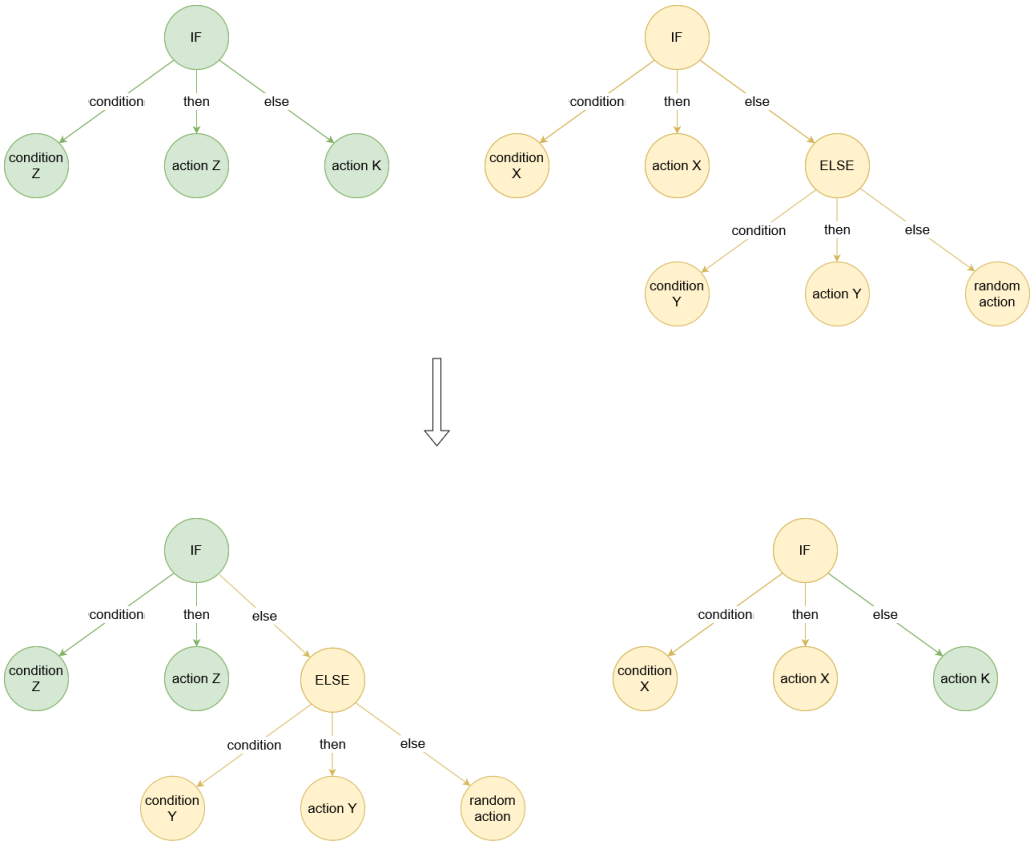


Figure 4.2: Cross-over - Replace one node of an individual, including its subtree with a node and a subtree of another individual. On the figure we can see the exchange of nodes between two strategies to produce two new individuals.

The evolution type used in this thesis is going to focus on Koza-style evolution of the GA [33]. The Koza-style genetic programming is an evolutionary computation technique that allows the evolution of tree-like structures (such as the Extended Backus–Naur form) by following their rules and object types. An example of how the GA processes are applied on strategies can be seen on Figures 4.2 and 4.3. For crossover we can see how the subtrees are exchanged between two strategies with different colours. Note that a single node is also a subtree. We can see that both subtrees return an action in every case. Similarly for mutation, the subtree (node "action X") has been replaced by a subtree that returns an action. Further details about the processes can be seen in the captions.

Overall, the benefit of a strategy over a sequence is that it will be possible to perform multiple runs per strategy, while still producing new test sequence on each run. 'in order to evaluate the quality of the sequences generated by the startegy as compared to directly generated sequences, we compare the results with work by Rauf and Anwar [30] on the SUT notepad.

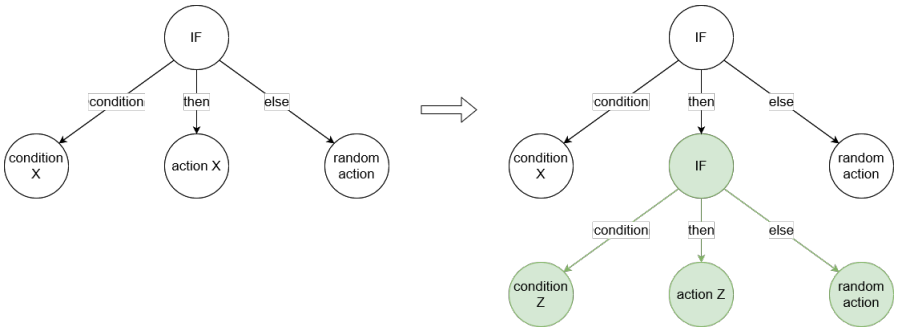


Figure 4.3: Mutation - Replace one node of an individual, including its subtree with a randomly generated tree with the same return type. On the figure we can see how action X has mutated to a subtree which returns a different action under condition Z

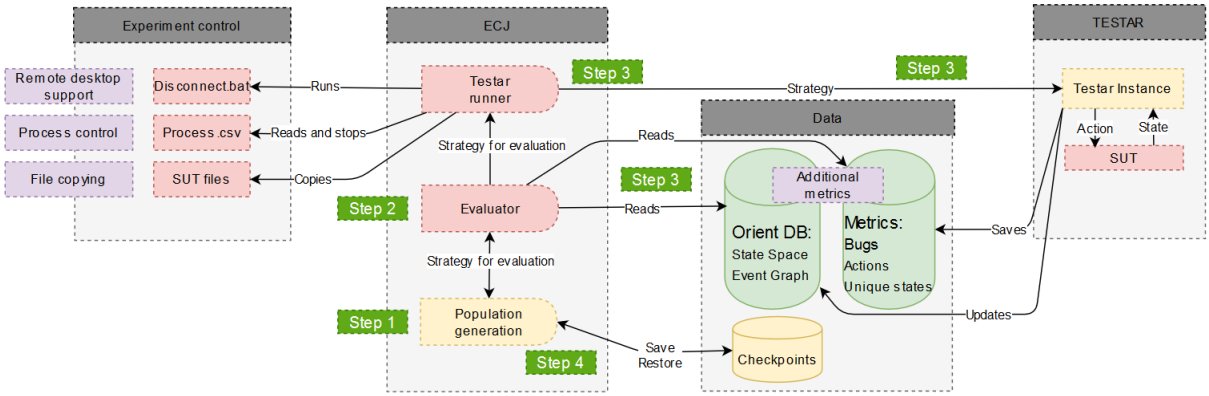


Figure 4.4: Architecture of the implementation. The modified parts in this thesis are coloured in either red or green. The red parts indicate modified or added code, while the green ones indicate added metrics. The bright green labels are mappings to step by step description of the picture. The purple labels map the added components to the text. The yellow parts are not modified during this thesis.

## 4.2 Architecture

In Figure 4.4 the architecture of the experiment set-up can be seen. It shows the interaction between the ECJ GA implementation and the testing tool TESTAR. The process is as follows for each generation.

**Step 1** A population of strategies is generated in the ECJ Population generation entity. The population is either fully randomly generated if this is the first generation, or is partially randomly generated while including the best individuals and the crossover from the previous generation.

**Step 2** The generated strategies are sent to the evaluator.

**Step 3** Each strategy is sent to the TESTAR runner class that starts a TESTAR process that performs an experiment on the SUT with the specified number of actions by following the strategy. After finishing the testing, the metrics gathered are saved in a file, indicated by a green cylinder in the picture. A state model is also generated in a Database. The evaluator performs the required calculations to evaluate the individual.

**Step 4** After repeating Step 2 and 3 for each strategy, the evaluation results are sent to the Population generation.

**Step 5** After evaluation if the maximum number of generations has not been reached, a selection process takes place. The selection approaches are Tournament Selection and Elitism. Then the process repeats itself from Step 1.

The framework also supports Java checkpoints. These checkpoints allow the user to restore the state of a running Java program back to a certain point in time by implementing the Java serializable interface. In this case, the checkpoints are taken at the end of each generation.

As an addition to the previous framework the functionality was extended. Further information about why each component was added can be seen in the descriptions of the components. Here we will explain each component that we added in order to improve and stabilize the testing framework:

**Process control** As described by Theuws [37] there were many issues with the experiment. He had to perform numerous restores on the virtual machine where the experiment was performed. During this research the main issue was pinpointed. As TESTAR would sometimes not finish on time, or alltogether pause multiple instances of TESTAR would be run. This in turn would break the system, as all the instances of TESTAR try to take control. In order to mitigate that both a process killer functionality was added and the timer of TESTAR was used. This resulted in a smooth execution of the experiments without overlapping TESTAR instances. This would require one to inspect in depth the related processes for each SUT in order to be able to stop them through different properties of the processes.

**Remote desktop support** A major issue encountered in this experiment was that the experiments would not work on disconnected remote desktops. After a while the issue was pinpointed to be due to a lock screen when one has disconnected from the remote desktop. A solution was found by disconnecting through a bat-file. This would require a shortcut to the bat-file administrator rights and removing the windows 10 notifications.

**Additional metrics** As already mentioned, the previous framework used the unique states for calculating the fitness value. Currently the framework has been extended to support all the available metrics from 2.3.1. The functionality to support the orientDB [5] as well as make queries through it was also added.

**File copying** Another issue that had to be bypassed was SUT that required an internet authorization. In order to bypass that we preconfigure the SUTs by injecting a configuration file.

## 4.3 Fitness functions

The fitness functions of this experiment are evaluating a given strategy by running this strategy a fixed number of times, specified in each experiment. Then each of the runs is being evaluated and the mean of the fitness of each of those runs is the fitness value of the strategy. The reason for having more than 1 runs is due to the fact that this will decrease the effect of the randomness of individual strategy executions. The fitness functions in this thesis are going to focus on a minimizing problem. Therefore they will try to reach a value as closely as possible to 0.

The previous experiment [37] conducted with ECJ and TESTAR has been minimizing the fitness function, $f_0 = 1/uniqueStates$, where $uniqueStates$ is the total number of unique states in the current run and the goal is to get $f$ as close to 0 as possible.

Although the function does focus on finding a strategy that will reach as many unique states as possible, an automated testing should focus on more than just exploring a high number of new states. Therefore, the fitness functions developed in this thesis are introducing different valuable information to be used as evaluation criteria. The fitness functions are:

**Curve steepness** The fitness function focuses on finding strategies that introduce new actions more often in smaller sub-sequences.

**State model coverage** The fitness function focuses on achieving a high state model coverage.

**Bug finder** The fitness function focuses on finding bugs. In this experiment, we are going to look for SUT with existing bug reports in order to determine whether we can successfully find those bugs by using an older version of the SUT.

### Fitness function 1: Curve steepness

Software testing in the industry generally needs to be as fast and efficient as possible. As mentioned before, by focusing on the total number of unique states, one does not take into account that a sequence can achieve a higher number of unique states in less actions. Therefore, to tackle this, we propose a fitness function about curve steepness. The function focuses on measuring how fast the current sequences find new unique states. However, as it is unreasonable to expect to generate a sequence which generates new states throughout the whole process, we focus on finding smaller consecutive sequences into the bigger one. Moreover, the sequences rarely introduce new states with each action. They generally introduce new states in a small number of actions which is the reason for adding a window size parameter. Generally a consecutive sequence is a sequence that introduces new states after some number of actions, where the maximum number of actions is the window size. Finally, to prioritize bigger sequences, over smaller separate sequences, which introduce the same number of unique states, an extraction parameter from the length of each sequence has been added. This function has been inspired by the notion of combinatorics in [8]. They look into analyzing the sequence of actions instead of analyzing resulting metrics. An example calculation with window size of 3 has been given in Figure 4.5:

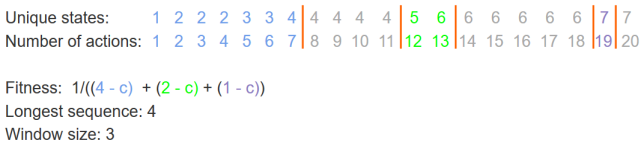


Figure 4.5: Curve steepness fitness function example. On the Figure 3 we can see a sequence of 20 actions. We can see 3 smaller sub-sequences, which continue to introduce new states in the given window size. They are indicated in different colours. The calculation fitness happens by extracting the parameter $c$ out of each sub-sequence length and minimizing the reverse of that sum.

#### Fitness function 2: State model abstract states

Exploring new unique states does not take into account the transition between states. For example, if a sequences goes from state X to state Y, and from state Z to state Y, it will not reward the latter one. However, it may be the case that the transition between states introduces new bugs or behaves differently than intended. Therefore, we would like to explore as many unique transition between states as possible. To evaluate that, the state model is used. As already discussed in Section 2.1.2, the state model looks into both the action transitions and the unique states. In TESTAR, the OrientDB technology is used. We use the fitness function:

$$f = 1/(AbstractStates)$$

where *AbstractStates* is the Abstract States in the state model.

This fitness function is similar to the one used in [30]. As that research generates sequences, we will compare the strategy and sequence generation.

#### Fitness function 3: Bug finder

The notion of software testing is often related to finding bugs in a software. However, the current implementation of unique states does not reward a sequence that finds bugs. Thus, we propose a fitness function that has been improved to drastically reward the bug-finding strategies:

$$f = 1/((uniqueStates) + bugFound * sequenceLength))$$

where *uniqueStates* is the total number of unique states , the *sequenceLength* is the maximum (intended) sequence length of the run and *bugFound* is either 1 or 0 depending on whether a bug has been found. In this thesis, the definition of a bug will be an existing bug report for a specific GUI. The bugs that we are looking for will be described in the specific experiments in Subsection 5.3.

One could argue that the reward for finding a bug in this thesis is too large. However, the current notion of strategy is a bit more abstract than the sequence finding one. This makes it a bit uncertain if the strategy will be effective in reproducing bug finding. Therefore, a big reward is given to investigate this.

### RQ1.1: What kind of evaluation criteria can be used in the strategy-based GA approach?

The first sub-question is answered by the research we performed and introducing the three new fitness functions. Now we would need to answer the next two sub-questions. This is done through the experiments discussed in the next chapter.

# Chapter 5

# Experiment set-up

The current research is going to perform four types of experiments. A short summary of each experiment is given below:

**Curve steepness experiment** The first experiment uses the fitness function for curve steepness which has been described in Subsection 4.3. It is performed on Notepad and VLC and it uses the same parameters as the experiment performed by Theuws[37]. The goal is to see whether we can direct the fitness function to achieve different testing goals, such as more valuable and shorter sequences, which is better for the industry due to the shorter execution time.

**State model experiment** The experiment focuses on evaluating the sequences through the state model of TESTAR [29]. The goal here is to compare the strategy approach as closely as possible to an already existing experiment which uses sequences. [30]

**Bug finding experiment** The third experiment aims to show us whether we are able to find bugs by using the notion of strategy. The experiment will show us if a strategy will be able to focus on finding a bug more often than the random approach.

**Long experiment** Due to time constraints, the previous research was limited in how much evaluations per strategy and number of generations it could achieve. In order to further explore whether the GA approach improves with time, the remaining time of the research has been focused on this experiment. It is evaluating both the original fitness function used by Theuws[37] and the curve steepness introduced in the current research.

An overview table of the first three GA experiments is given in Table:

| Fitness function | Curve steepness | State model abstract states | Bug finder |
|---|---|---|---|
| SUT | Notepad, VLC | Notepad, VLC | PAINT.NET, Cyberduck |
| Number of actions | 100 | 100 | 100 |
| Evaluations per strategy | 5 | 2 | 5 |
| Population size | 100 | 100 | 100 |
| Generations | 10 | 10 | 10 |
| Expected time per SUT: | 70 hours | 140 hours | 70 hours |

Table 5.1: Summary of GA experiments

For the rest of this chapter we are going to address the random action selection mechanism as the Random Action Strategy(RAS) for convenience. The experiments were performed on a remote desktop machine. The information about the system is given in Appendix A. The strategies resulting from the experiments are given in Appendix B.

## 5.1 Curve steepness experiment

The first experiment is going to be performed by using the fitness function of curve steepness that has been discussed in Subsection 4.3. The experiment mimics the experiment performed by Theuws [37] . It evaluates the fitness function on the SUTs Notepad and VLC in order to see whether changing the fitness function will help us navigate it towards a different goal. In this case it is trying to save time in order to perform more valuable tests in shorter time.

### 5.1.1 Technical details

An issue that needs to be overcome here is a pop-up when testing the VLC SUT. If one is performing the experiment on a remote desktop, that pop-up may hide under the start menu due to resolution changes when disconnecting. The work-around is to always close this pop-up when it appears.

### 5.1.2 GA set-up

**SUT** Notepad (The notepad version is related to the Windows version given in Appendix B) , VLC (version 3.0.11)

**Number of actions per strategy** 100 actions have been chosen to be performed per evaluations. This is chosen due to being used in the previous research and due to time constraints.

**Evaluations per strategy** 5 evaluations have been chosen per strategy due to time constraints.

**Population size** 100 individuals have been selected per generation, as this is the population size of the previous researches.

**Generations** 10 generations have been selected due to time constraints.

**Elitism** 10 of the best individuals per generation are carried to the next generation.

**Mutation rate** 5% this is the usual parameter used for mutation in most of the GA approaches.

**Tournament selection** 70% of the population per generation takes part of the generation. We have chosen a high value as this will create a greater chance that fitter individuals will be part of the genetic processes.

**Run time per experiment** Around 4 and a half days

### 5.1.3 Random set-up

As part of the evaluation of the strategies, we have generated 1000 random strategies for both SUTs and evaluated each five times in order to select the best one for each SUT. For the rest of the chapter this strategy is going to be called Best Random Strategy(BRS). The fitness function is described in Subsection 4.3. The reason for choosing this number of individuals and evaluations is to do similar number of evaluations as compared to the GA, although we still do 10% less evaluations with the GA due to the Elitism selection process.

### 5.1.4 Evaluation

For the evaluation we are going to keep following the model proposed by Theuws. In order to determine how good the GA strategy behaves we are going to compare it to two other action selection methods: RAS and BRS. The comparison to RAS will give an overview on how well a strategy performs, while the comparison to BRS will help us determine whether GA is a good way to develop strategies. The rest of this subsection will explain the evaluation set-up and the statistical methods used to compare the results of the evaluation.

### Evaluations experiments description

As already discussed we would like to measure whether the fittest GA individuals perform better than the RAS and the BRS. In order to do so we have selected the three strategies with the best fitness value from the GA. They are named as the first GA strategy, the second GA strategy and the third GA strategy, according to how good they performed during the GA evaluations. We are going to perform two evaluations. One of them is going to run each strategy 1000 times with runs of length 100 and the other is going to run them 30 times with runs of length 500 and compare the results of the fitness function on those runs.

### Statistical methods

**General assumptions:**

We are going to be comparing five groups of results at a time. The results of the previous researches are not normally distributed and we so assume that the results of this thesis will also not be normally distributed. Therefore, we must use non-parametric test.

Another assumption we make is that our data is consisting of continuous variable. Continuous variables are numeric variables that have an infinite number of values between any two values. As we are dealing with fitness functions which will both have a maximum and minimum results (best and worst fitness function). In order to perform tests we are going to treat each fitness function as a continuous variable between those two values.

We will also assume that very specific runs should not influence our research, as we are looking for a balanced performance instead of a very specific one. Therefore we are going to remove outliers from the data. An outlier is a value that significantly differs from the other values in the group.

**Test:**

By following the assumptions and requirements we can pinpoint the best test for our research. It is the Kruskal-Wallis H-test [26]. The test is also known as one-way Anova on ranks. It is used to determine whether there are statistically significant differences between two or more groups of an independent variable on a continuous dependent variable. The test result is a value which helps us determine whether we can reject the null hypothesis. The null hypothesis if usually the following: “There is no statistical difference between the measured groups”. In our case the null hypothesis will be: “There is no difference between the performance of the random action selection mechanism and any of the strategies”.

In order to user the Kruskal-Wallis H-test we need to make sure that the following assumptions hold:

**The dependent variable should be measured at the ordinal or continuous level.** In our case this is the fitness function value, which we already discussed as continuous.

**Your independent variable should consist of two or more categorical, independent groups.** In our case there are 5 categorical independent groups. The randomly chosen strategy, the random action selection mechanism and the 3 strategies from the GA.

**You should have independence of observations, which means that there is no relationship between the observations in each group or between the groups themselves.** As our groups are based solely on the chosen strategy, there is no overlap between them.

The test works as follows. We assume a null hypothesis as described above. The test produces a value. The result is compared to a $p$ value and if that value is smaller than $p$ we reject the null hypothesis. In this research we are going to use the value of $p = 0.05$ as this is the norm in statistical research.

### 5.1.5 Results and conclusions

**Notepad GA experiment**

Here we will discuss the results of 1000 runs of length of 100 actions in the Notepad SUT. The evaluation was performed on all of the specified strategies. Please note that for showing the results we use the inverse of the fitness function. In order to see whether there is improvement over the generations during the GA we are going to analyze the mean values of the Elite strategies over the generation. As there is a high number of random strategies per generation, the Elite strategies will give us a better indication of the improvement during the generations. These values can be seen in Figure 5.1. As we can see the improvement seems to be lessened with the higher number of generations. At the end, the generations 7 and 10 even fail to produce a new elite strategy. However, as there still seems to be a steady improvement, it is possible that more generations will yield better strategies.

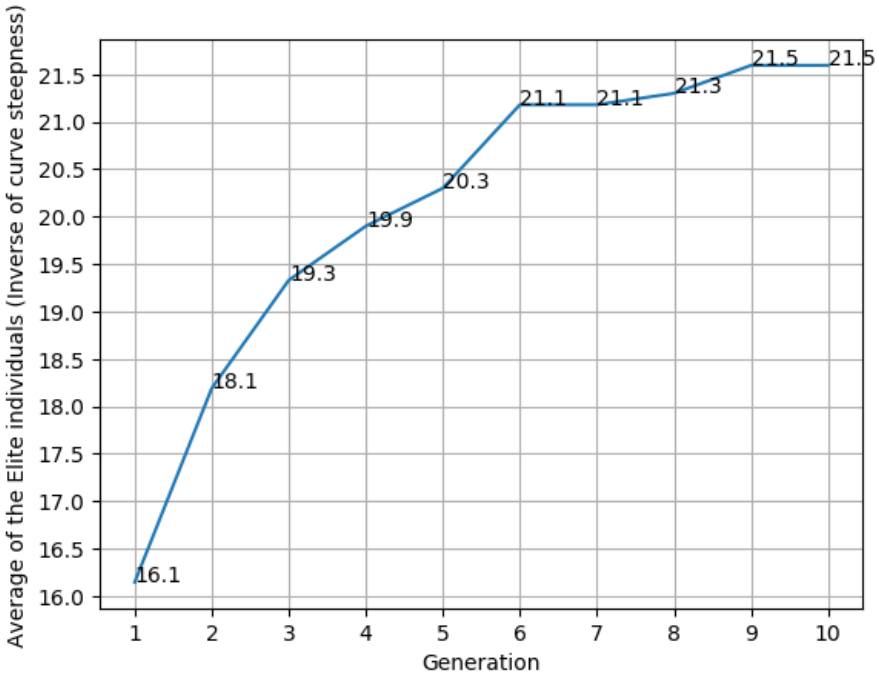


Figure 5.1: Mean values of the Elite strategies over the generations of Notepad curve steepness.

### Notepad 1000 runs of length 100

A boxplot of the results is given in Figure 5.2. Please note that for showing the results we use the inverse of the fitness function. This means that the higher the result, the better. After the analysis a conclusion was made that this helps in clarification of the results.

The results of the Kruskal-Wallis H-test is $5.122928823142474e-105 < 0.05$ which means that we reject the null hypothesis. Further research and tweaking of the experiment also showed that none of the strategies resulted in a similar distribution. This means that we can use the mean values to determine whether a strategy performed better than another. The mean values are given in Table 5.2. The tweaking of the experiment is running the test with different tuples of the results of the different strategies.

From the boxplot we can see that all the strategies have a similar variance in the results, although the Third GA strategy and the RAS seem to have more exceptionally good outliers. Nevertheless, due to the rejection of the null hypothesis and the mean values we can conclude that the first GA strategy performed best followed by BRS, RAS, the third GA strategy and the second GA strategy:

- As the first GA strategy and the BRS outperformed the RAS we can conclude that the strategy approach is better than the random action selection mechanism, even thought the

| | Mean (average) |
|---|---|
| First GA strategy | 25.7 |
| Second GA strategy | 18.3 |
| Third GA strategy | 19.5 |
| RAS | 20.2 |
| BRS | 24.5 |

Table 5.2: Table of the mean values of the results of Notepad 1000 runs of length 100.

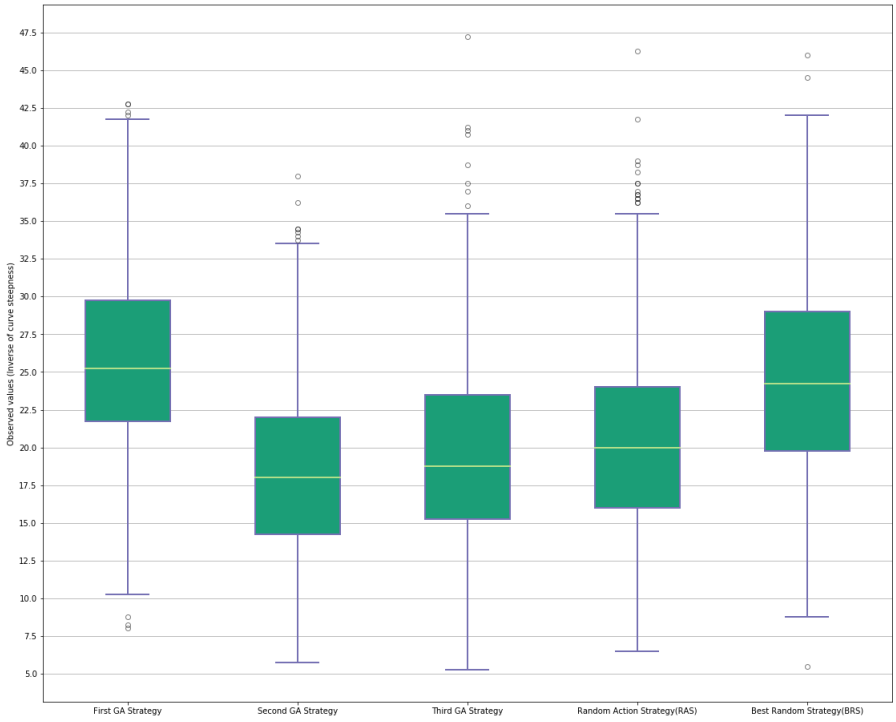


Figure 5.2: Boxplot of the Notepad 1000 runs of length 100.

second and third GA strategy performed worse than the RAS.

- Since the third GA strategy outperforms the second one, we can conclude that the GA needs more evaluations during the experiment.

Due to the experiment results we can make the following conclusions:

- The GA outperforms randomly selecting a given strategy (BRS). This is seen due to the first outperforming the BRS.
- The strategy approach, given a good enough strategy can outperform the RAS.
- More evaluations are required for the GA experiments. This is seen as the third GA strategy outperforms the second GA strategy which was not the case during the GA experiments.

### Notepad 30 runs of length 500

A boxplot of the results is given in Figure 5.3. The results of the Kruskal-Wallis H-test is $5.122928823142474e-105 < 0.05$ which means that we reject the null hypothesis. Further research and tweaking of the experiment also showed that the second GA strategy and the BRS have similar distribution, while the rest are different. The mean values are given in Table 5.3.

| | Mean (average) |
|---|---|
| First GA strategy | 70.1 |
| Second GA strategy | 60.7 |
| Third GA strategy | 57.0 |
| RAS | 61.1 |
| BRS | 64 |

Table 5.3: Table of the mean values of the results of Notepad 30 runs of length 500.

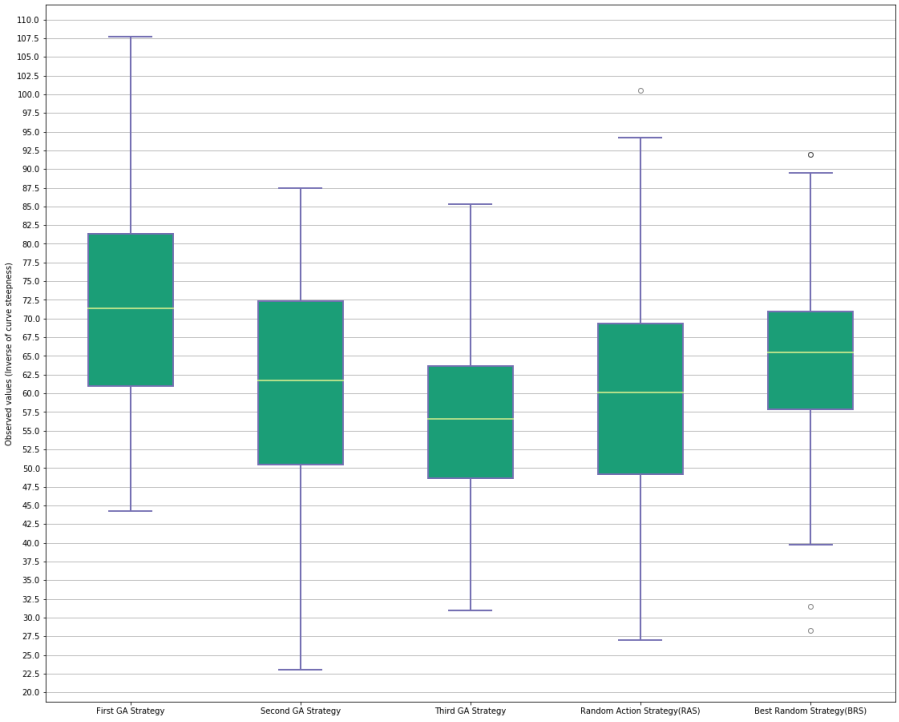


Figure 5.3: Boxplot of the Notepad 30 runs of length 500.

The results of the evaluation with length 500 is quite similar to the previous one. The only difference is that the second GA strategy in this case performs better than the the third one. Surprisingly, even though the second GA strategy and the BRS share similar distribution according to the test, the BRS outperforms the RAS, while the other does not. This lessens the strength of the argument that BRS outperforms the RAS, even though the result values point otherwise. The conclusions we can make from the experiment are the following:

- The GA outperforms randomly selecting a given strategy (BRS). This is seen due to the first outperforming the BRS.
- The strategy approach, given a good enough strategy can outperform the RAS.

In this experiment we can not argue that more evaluations are needed, as the first, second and third GA strategy perform in order. However, seeing as the previous experiment (length 1000) has a higher number of evaluations, we can still argue that more evaluations during the GA experiment are needed.

### VLC GA experiment

Here we will discuss the results of 1000 runs of length of 100 actions in the VLC SUT. The evaluation was performed on all of the specified strategies. Unfortunately the statistics of the GA experiment for VLC curve steepness have been lost during testing. Please note that for showing the results we use the inverse of the fitness function.

### VLC 1000 runs of length 100

A boxplot of the results is given in Figure 5.4. This means that the higher the result, the better. After the analysis a conclusion was made that this helps in clarification of the results.

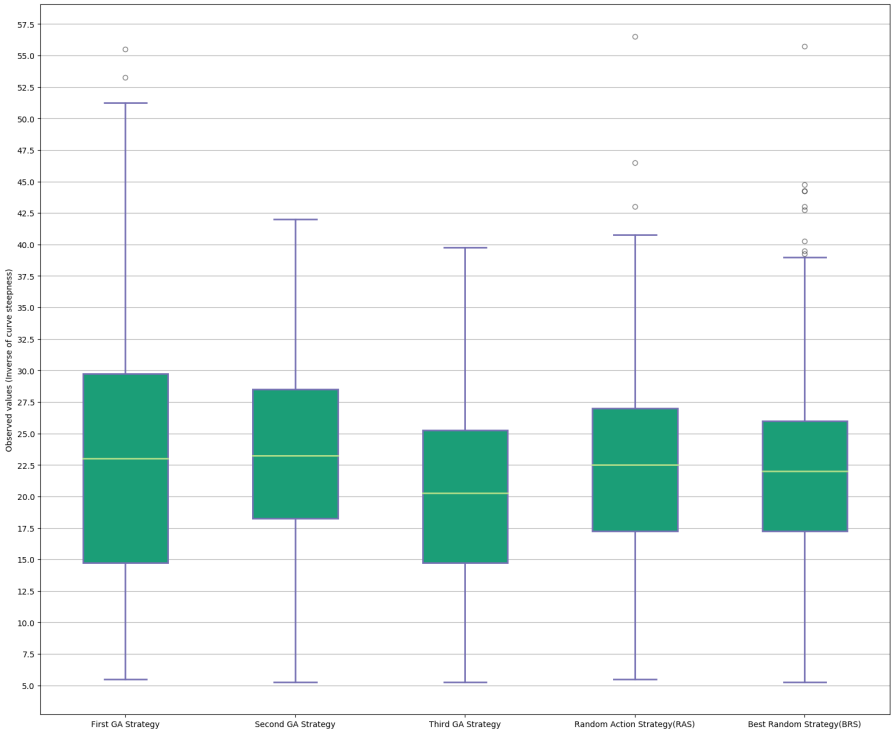


Figure 5.4: Boxplot of the VLC 1000 runs of length 100.

The results of the Kruskal-Wallis H-test is $2.6791887376268187e-15 < 0.05$ which means that we reject the null hypothesis. After some tweaking of the test, the results showed that only the best GA strategy and the RAS results were of a similar distribution. The averages of the results are given in Table 5.4.

From the boxplot we can see that the first GA strategy has more variation in the results, as compared to the rest which were more focused around the median. We can make the following conclusions for the results:

| | Mean (average) |
|---|---|
| First GA strategy | 22.5 |
| Second GA strategy | 23.1 |
| Third GA strategy | 20 |
| RAS | 22 |
| BRS | 21.5 |

Table 5.4: Table of the mean values of the results of VLC 1000 runs of length 100.

- The second GA strategy is of different distribution than each of the rest due to the rejection of the null hypothesis. As it has a higher median, we can conclude that it outperforms all the other strategies.
- The first GA strategy and the RAS are of similar distribution as the null hypothesis holds. Although the first GA strategy slightly outperforms the RAS due to the higher median, it has more varying results. We can not conclude whether one is better than the other.
- The third GA strategy and the randomly selected best strategy are of different distributions due to the null hypothesis and seem to have lower values as compared to the rest of the strategies.

Due to the experiment results we can make the following general conclusions:

- The GA outperforms randomly selecting a given strategy (BRS). This is seen due to the first and second strategy both outperforming the BRS.
- The strategy approach, given a good enough strategy can outperform the RAS.
- More evaluations are required for the GA experiments. This is seen as the second GA strategy outperforms the first GA strategy which was not the case during the GA experiments.

### VLC 30 runs of length 500

This experiment was performed on the five strategies by performing 30 runs of length 500 for each on the SUT VLC. The boxplot of the results is given in Figure 5.5. The means are given in Table 5.5

The results of the Kruskal-Wallis test are $0.08868415273442869 > 0.05$. This means that we can not reject the null hypothesis, and all the groups are of similar distribution. The results of the test are surprising as the results of each group look quite different in the figure. This can also explain why the result of the test is quite close to the $p$ value. Due to the surprising results of the test a couple of more pairing tests were performed on different groupings. The results showed that all the GA strategy are quite similar to each other. It also showed that the third GA strategy is quite similar to the RAS and BRS, while the other two are not. This could perhaps explain to some extent why the sum all of them together depicts them as similar distributions. Nevertheless, although not rejecting the null hypothesis prevents us from making definite conclusions, we can clearly see the higher values coming from the GA strategies.

| | Mean (average) |
|---|---|
| First GA strategy | 60.9 |
| Second GA strategy | 48.9 |
| Third GA strategy | 52.8 |
| RAS | 48.1 |
| BRS | 43.5 |

Table 5.5: Table of the mean values of the results of VLC 30 runs of length 500.

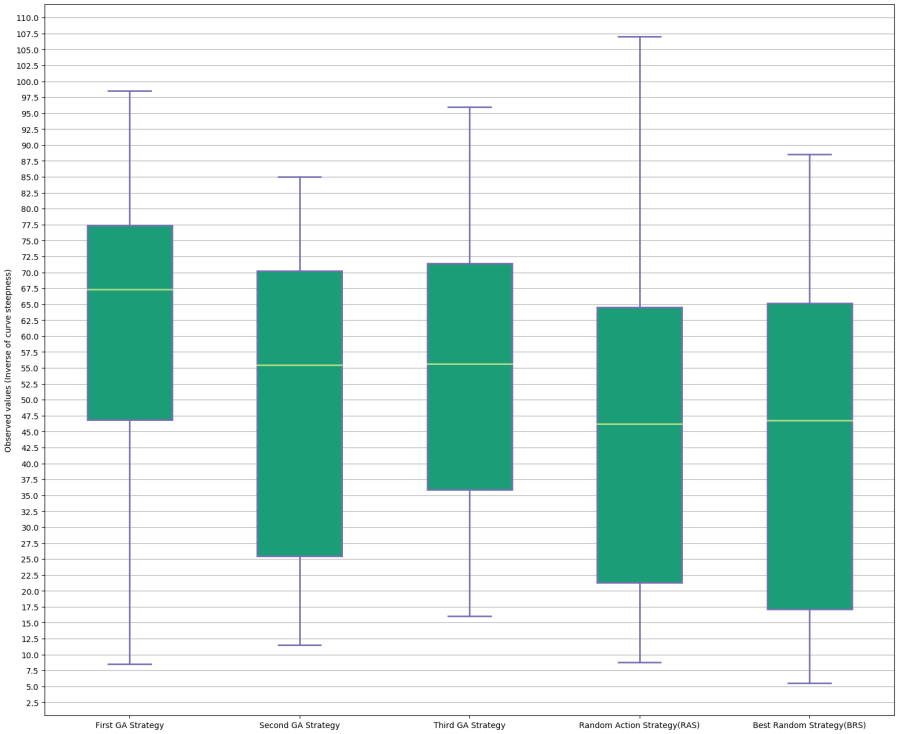


Figure 5.5: Boxplot of the VLC 30 runs of length 500.

## 5.2 State model experiment

The second experiment is going to be using the state model of TESTAR which has been discussed in Subsection 2.1.2. The goal of the experiment will be to cover as many abstract states as possible. For the evaluation of the results we are going to check how many of the states was covered by the GA strategy as opposed to the RAS. We are also going to compare our results to an already existing approach dealing with sequences [30]. This will hopefully give us some indication about how the strategy compares to specific sequences, although dealing with a more complex model will most likely result in less coverage.

### 5.2.1 Technical details

The VLC issue that was present in the first experiment was not impacting this one. Most likely this is due to the slower execution of this experiment which was not causing VLC to override it's settings.

### 5.2.2 GA set-up

**SUT** Notepad (The notepad version is related to the Windows version given in Appendix B) , VLC (version 3.0.11)

**Number of actions per strategy** 100 actions have been chosen to be performed per evaluations. This is chosen due to being used in the previous research and due to time constraints.

**Evaluations per strategy** 2 evaluations have been chosen per strategy due to time constraints.

**Population size** 100 individuals have been selected per generation, as this is the population size of the previous researches.

**Mutation rate** 5% this is the usual parameter used for mutation in most of the GA approaches.

**Generations** 10 generations have been selected due to time constraints.

**Elitism** 10 of the best individuals per generation are carried to the next generation.

**Tournament selection** 70% of the population per generation takes part of the generation.

**Run time per experiment** Between 7 and 8 days per SUT

The set-up of the GA part of the experiment is similar to experiment one. However, due to the execution time increase while using the state model we had to lower the number of evaluations per strategy. The current GA set-up took between 7 and 8 days to finish. It resulted in a state model with 1300 abstract states for notepad and 6087 abstract states for VLC.

### 5.2.3 Evaluation

From the results of the current experiment we would like to see the coverage of the state model that strategies manage to achieve. Ideally it would be good to compare the BRS to the GA generated strategies, however as whether GA outperforms the random generation of strategies is part of experiment 1 and due to the high execution and therefore generation time of the state model based executions we are not going to generate a BRS in the current experiment. In the current experiment we are going to evaluate 4 strategies: RAS and the three fittest individuals from the GA. Each of them is going to perform 200 runs of length 100 and 50 runs of length 500. Please note that for both types of run sizes the strategies will generate a separate state model. Considering 4 strategies with 2 types of run sizes results in 8 separate state models per SUT. We will evaluate how much of the abstract states in the combined state model are covered by each smaller state model. Please note that these state models are not complete: The ones resulting

from the GA are a combination of around 2000 executions, however it could be the case that there are states still not covered by those runs. However, there is no complete state model on any of the SUTs at the current date.

### 5.2.4 Results details

**Notepad GA experiment**

Similarly to the first experiment we take the Elite strategies of each generation in order to see whether the GA strategies improve with time. The mean values are seen in Figure 5.6. Similar results to the first experiment are found, as the values keep increasing steadily over the generations, even though no new elite strategies are found during the last one. Overall 1308 abstract states were found while performing the experiments. After all the evaluation experiments that number of those states increased up to 2340.

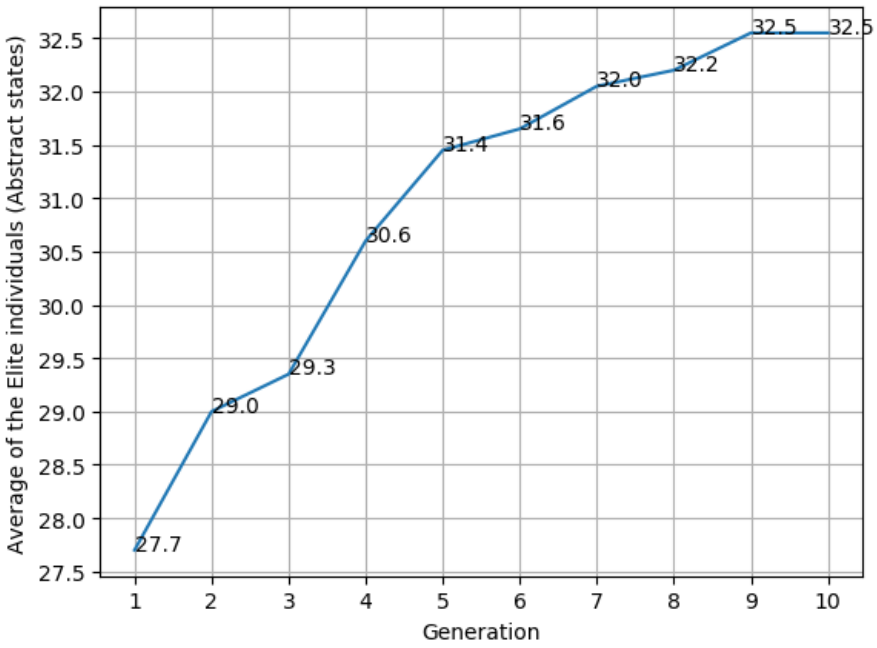


Figure 5.6: Mean values of the Elite strategies over the generations of Notepad state model abstract states.

| Number of runs | First GA strategy | Second GA strategy | Third GA strategy | RAS |
|---|---|---|---|---|
| 40 | 244 (10%) | 329 (14%) | 223 (9%) | 248 (10%) |
| 80 | 375 (16%) | 441 (18%) | 288 (12%) | 329 (14%) |
| 120 | 437 (18%) | 581 (24%) | 326 (13%) | 422 (18%) |
| 160 | 511 (21%) | 672 (28%) | 372 (15%) | 496 (21%) |
| 200 | 542 (23%) | 726 (31%) | 429 (18%) | 536 (22%) |

Table 5.6: Notepad 200 runs of length 100 abstract state coverage.

### Notepad 200 runs of length 100

As we already discussed we are aiming to compare these results to the results achieve by Rauf et al. [30]. The authors were able to achieve a coverage of more than 80% after 300 generations. Due to the running times, it is not possible to achieve 300 generations with GA in our framework as it would take supposedly more than half a year. We are also dealing with a more complex model, which would definitely cause the results to achieve lower coverage than on a simpler one. The results of 200 runs of length 100 are given in Table 5.6 and Figure 5.7. All of the runs per strategy were being performed on the same model, meaning that the 200 runs were building the same model.

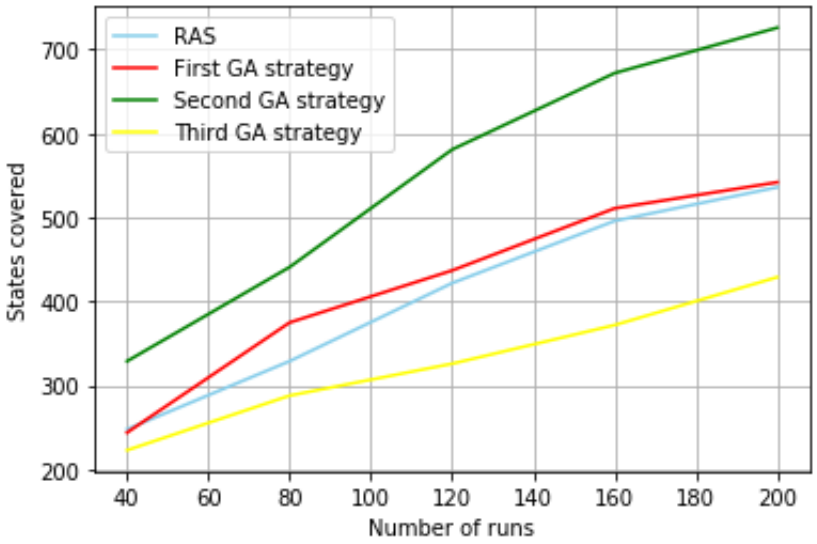


Figure 5.7: Notepad 200 runs of length 100 abstract state coverage.

From the results we can see that were not able to achieve the coverage described by the other research. However, we see that the number of abstract states keeps increasing with the larger number of runs and it would most likely keep increasing further. However, as the experiment already took over 55 hours evaluating each strategy it will not be possible to perform more evaluations during this thesis. We can see that the first GA strategy and the RAS are performing similarly, but the second GA strategy significantly outperforms both of them(it manages to achieve almost 10% more coverage on the same number of runs). The big difference comes most likely from the low number of evaluations during the GA experiment. From the experiment we can conclude that the strategy approach outperforms the RAS and that we need a higher number of evaluations in order to achieve better strategies due to the big difference in the results of the first three GA strategies.

| Number of runs | First GA strategy | Second GA strategy | Third GA strategy | RAS |
|---|---|---|---|---|
| 10 | 262 (11%) | 417 (18%) | 254 (11%) | 254 (11%) |
| 20 | 367 (14%) | 650 (28%) | 429 (18%) | 404 (17%) |
| 30 | 427 (18%) | 746 (32%) | 513 (22%) | 495 (21%) |
| 40 | 480 (20%) | 864 (37%) | 513 (22%) | 625 (27%) |
| 50 | 526 (22%) | 915 (39%) | 588 (25%) | 668 (28%) |

Table 5.7: Notepad 50 runs of length 500 abstract state coverage.

### Notepad 50 runs of length 500

The results of the experiment can be seen in Table 5.7 and Figure 5.8. The second GA strategy, similarly to the previous experiment is performing significantly better than the other strategies. What is different in this experiment is that the first GA strategy underperforms compared to the others, which was not the case for the shorter runs. One possible reason for that is that the GA experiment was evaluated on runs of length 100, and the first GA strategy is simply not fit for longer runs.

The experiment performs 500 actions per run, meaning that in 40 runs it has performed 2000 actions. This is the total number of actions performed by each strategy in the experiment with runs of length 100. We can see that for each of the strategies, except the first GA strategy, longer runs tend to achieve a higher number of states, even if the total number of actions is the same. Perhaps increasing the number of actions for the GA experiment will yield better results, but that is not possible during the time frame of this experiment.

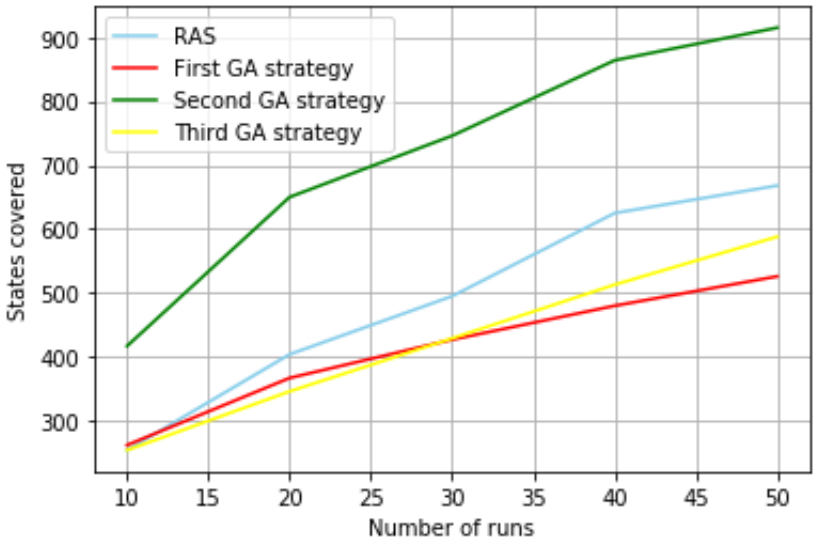


Figure 5.8: Notepad 50 runs of length 500 abstract state coverage.

### VLC GA experiment

We take the Elite strategies of each generation in order to see whether the GA strategies improve with time. The mean values are seen in Figure 5.9. This is the first experiment until now which does not seem to exhibit good improvement over all of the 10 generations. Most likely running more generations will not benefit this experiment, although the results could be caused by the low number of evaluations. After the GA experiment the model contained around 6500 abstract states. During the evaluations that number rose up to 10800, which would entail that there is still a lot of abstract states to explore in VLC.

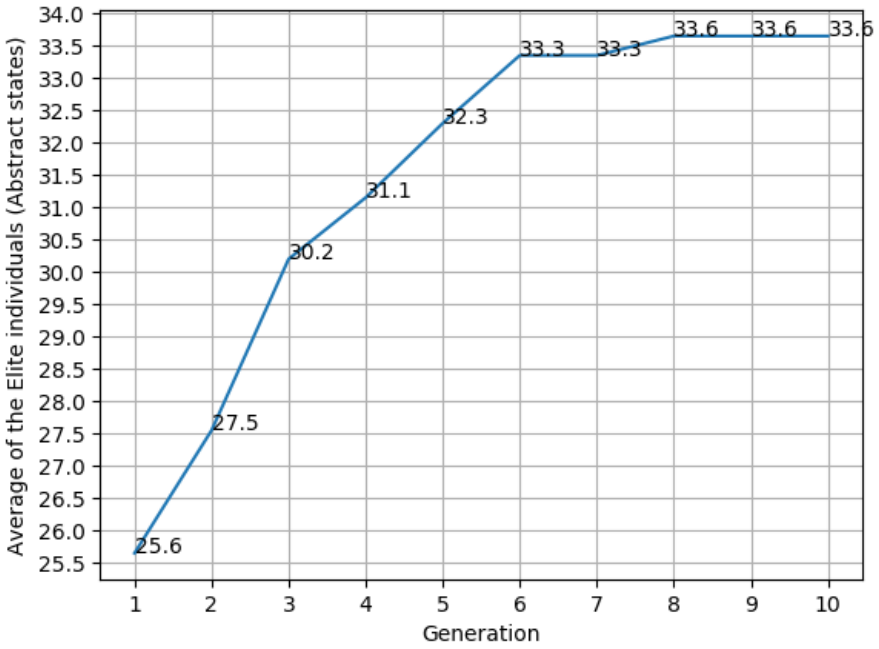


Figure 5.9: Mean values of the Elite strategies over the generations of VLC state model abstract states.

| Number of runs | First GA strategy | Second GA strategy | RAS |
|---|---|---|---|
| 40 | 507 (5%) | 498 (4%) | 606 (5%) |
| 80 | 968 (9%) | 1004 (9%) | 1069 (10%) |
| 120 | 1340 (12%) | 1528 (14%) | 1486 (13%) |
| 160 | 1722 (16%) | 1914 (18%) | 1926 (18%) |
| 200 | 1965 (18%) | 2319 (21%) | 2293 (21%) |

Table 5.8: VLC 200 runs of length 100 abstract state coverage.

### VLC 200 runs of length 100

As we already concluded in the previous experiments, the number of runs are not enough in order to achieve the high coverage described in previous research [30]. The VLC experiment also took longer time and we were not able to evaluate the third GA strategy on the SUT. The results are given in Table 5.8 and Figure 5.10.

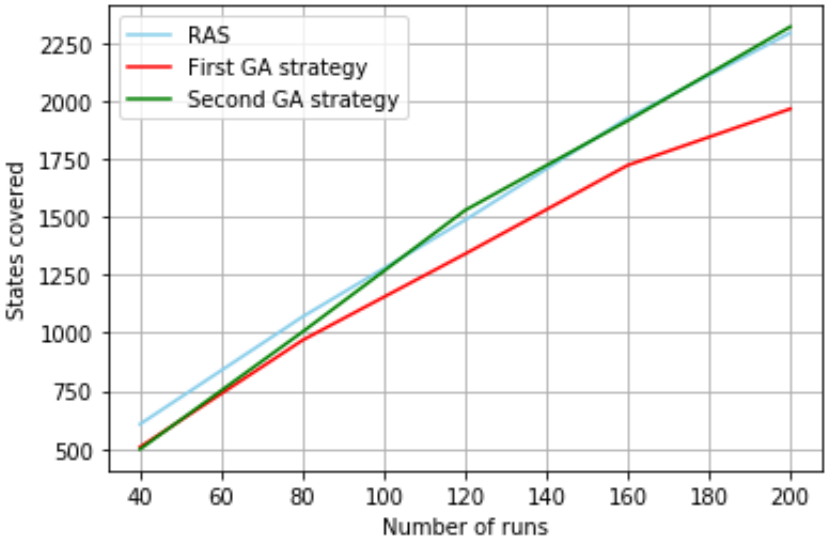


Figure 5.10: VLC 200 runs of length 100 abstract state coverage.

In the VLC experiment both the strategies and the RAS approach performed similarly. The coverage achieved is also similar to the Notepad experiment of the same size. Therefore, the only conclusion we can make here is that we need more number of evaluations, as the second GA strategy outperforms the first.

| Number of runs | First GA strategy | RAS |
|---|---|---|
| 10 | 638 (6%) | 277 (3%) |
| 20 | 1164 (11%) | 875 (8%) |
| 30 | 1507 (14%) | 1264 (11%) |
| 40 | 1952 (18%) | 1597 (15%) |
| 50 | 2400 (22%) | 1892 (18%) |

Table 5.9: VLC 50 runs of length 500 abstract state coverage.

#### VLC 50 runs of length 500

Unfortunately we were unable to finish the experiment with the second GA strategy. Thus we will be only evaluating the first GA strategy and the RAS. The results are given in Table 5.9 and Figure 5.11. Contrary to the shorter experiment, the first GA strategy seems to be performing much better than the RAS. We should also note that in the Notepad experiment with state model, the first GA strategy was performing worse on the longer experiment, while it does not seem to be the case here. In the Notepad variant we could also see that in most of the cases the same number of actions in total, but being executed by longer sequences was introducing more states than the shorter variant. In the current experiment this number (200 runs of length 100 and 40 runs of length 500) seems to be the same for the first GA strategy, while the RAS performs better on the shorter sequences.

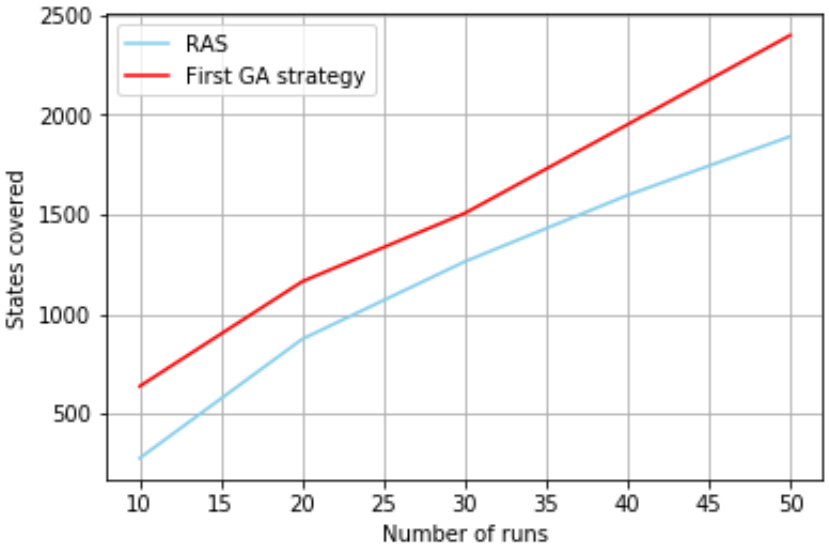


Figure 5.11: VLC 50 runs of length 500 abstract state coverage.

## 5.3 Bug finding experiment

The current experiment is going to follow the fitness function described in Section 4.3. The experiment will be performed on two different SUTs which have already existing bug reports. The experiment is going to help us answer the question whether a strategy can be focused towards finding specific bugs.

### 5.3.1 Technical details

#### Cyberduck

The software goal is to provide a library for your different storage platforms, such as Dropbox or Google drive. The current software exhibits the following bug. If there is a colon(:) in a file name the software will crash. Unfortunately, a colon in the filename is not allowed in Windows. A workaround was found by connecting the library to Google drive, where you can add colon to filenames. Once the crash occured, however, one would be disconnected from his google account. In order to reconnect again you would need to provide authentication by receiving a text message in your phone, a process that was impossible to automate through the TESTAR protocol. In order to bypass this, a configuration file with the existing connection to Google was saved on the computer, and was copied to the configuration folder of Cyberduck before each run.

In order to reproduce the bugs one would need to double click on the button showed in Figure 5.12[1]. The screen shown there is initial screen showed once you run Cyberduck with the Google drive configuration.

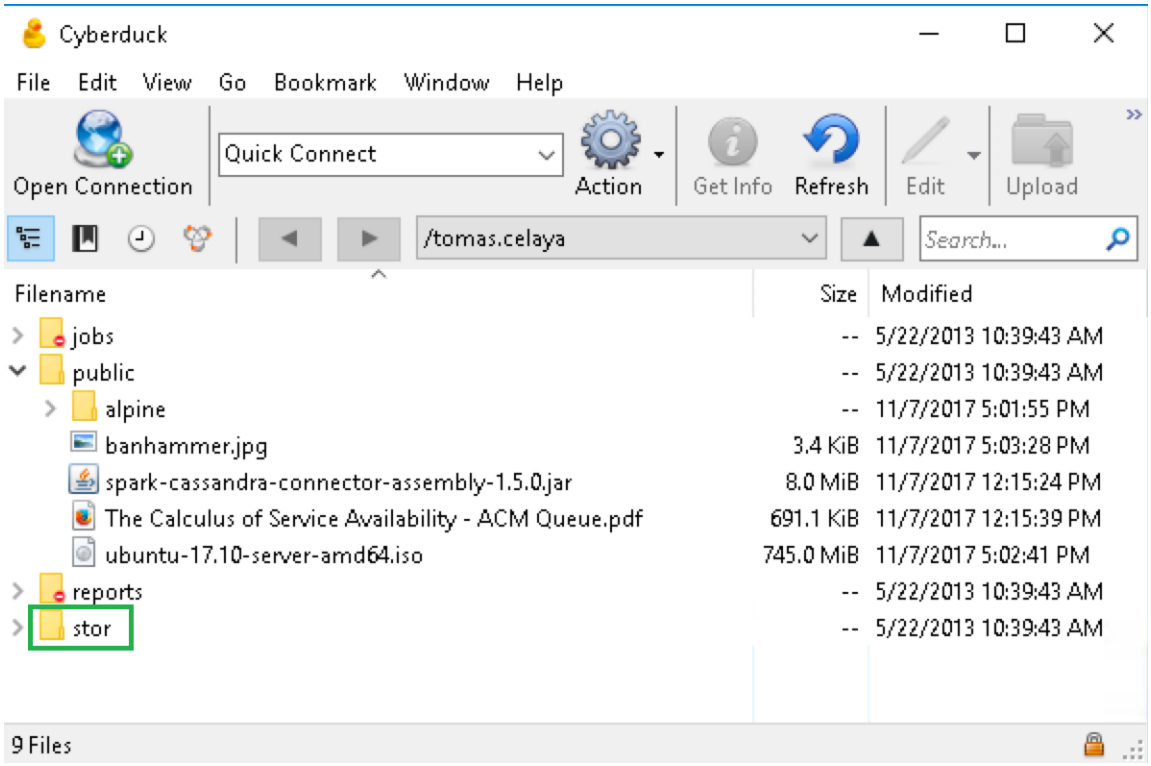


Figure 5.12: Cyberduck bug. Pressing the green button twice will result in showing the menu where the filename with semicolon is, thus triggering the bug. There is also a single click option on the arrow on the left, but that was not recognized as separate button by TESTAR.

#### Paint.net

In Figure we can see the initial screen of Paint.net. The bug which we wanted to reproduce was clicking on any options part of the green rectangle in Figure 5.13[6]. The bug occurs if the first action you did after clicking on the tools in the rectangle is pressing space.

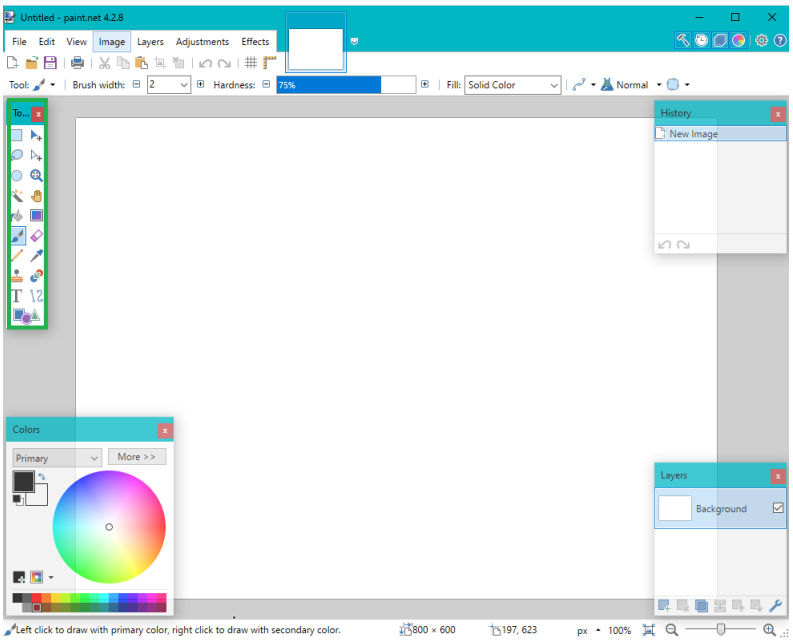


Figure 5.13: Paint.NET bug. Pressing any of the buttons surrounded by the green rectangle and immediately pressing space afterwards trigger the bug. Performing any action between them will cause the bug to not trigger.

### 5.3.2 GA set-up

**SUT** Cyberduck (version 7.4.1) , Paint.net (version 4.2.8)

**Number of actions per strategy** 100 actions have been chosen to be performed per evaluations. This is chosen due to being used in the previous research and due to time constraints.

**Evaluations per strategy** 5 evaluations have been chosen per strategy due to time constraints.

**Population size** 100 individuals have been selected per generation, as this is the population size of the previous researches.

**Generations** 10 generations have been selected due to time constraints.

**Elitism** 10 of the best individuals per generation are carried to the next generation.

**Mutation rate** 5% this is the usual parameter used for mutation in most of the GA approaches.

**Tournament selection** 70% of the population per generation takes part of the generation.

**Run time per experiment** Between 5 and 6 days for Cyberduck and 5 days for Paint.net.

The GA set-up is the same as the first experiment. However, it uses different fitness function and SUT. The start-up of Cyberduck was not successful every time. This was also the case if one just double clicked on the desktop icon. In those cases the framework had to wait the specified time, in this case around 8 minutes and start the software again. This resulted in a long running time of the experiment which was around 6 days.

### 5.3.3 Evaluation

The evaluation is going to perform 1000 runs of length 100 with the fittest strategy and see how many times it was able to find the said bug. In order to see whether the strategy was able to outperform the RAS in finding this bug, the same is going to be done with the RAS.

### Cyberduck

After running the GA experiment the bug was found 4 times in total. However, one of the strategies managed to find it twice. Therefore, that strategy was used for evaluation. Due to the bug being found quite rarely, the rest of the fitness function was discarded, while we only evaluated on whether the bug was found or not. The results were the following:

- The GA strategy found the bug 48 times out of 1000 runs. From the result we can conclude that the strategy managed to find the bug twice in five runs was the result of luck.
- The random-action selection found the bug 60 times of 1000 runs. It seems that random-action selection managed to find the said bug more often.

We can conclude here that we were not successful in biasing the strategy towards finding that specific bug. Even the random-action selection managed to outperform the strategy, although both of them found the bug quite rarely, so it is most likely the result of luck.

### Paint.net

Unfortunately the bug described for Paint.net was not found during the GA experiments. We still performed an evaluation of 800 runs with the RAS in order to see whether it would be possible to find the bug with it, but it also failed to find the bug. It is most likely too specific for the strategy approach. It would most likely be successful if you would only need to perform a key action after selecting one of the specified buttons, but the chance for that key to be space is perhaps too low for the size of the experiment we are conducting.

## 5.4 Long experiment

This experiment has been performed on two different fitness functions on the SUT. The goal of the experiment is to use the remaining time of the thesis in order to perform as long experiments as possible. The GA algorithm experiment set-up is similar to the curve steepness experiment. It was performed on the SUT Notepad and the evaluations per strategy were increased to 15.

### 5.4.1 Curve steepness long experiment

This experiment was performed on the curve steepness fitness function. It was able to achieve eight generations until the time to stop it. In order to evaluate the strategy from this experiment, we are going to compare it to the best strategy of the previous curve steepness experiment - namely the first GA strategy from there.

#### GA Set-up

**SUT** Notepad (The notepad version is related to the Windows version given in Appendix B)

**Number of actions per strategy** 100 actions have been chosen to be performed per evaluations. This is chosen due to being used in the previous research and due to time constraints.

**Evaluations per strategy** 15 evaluations have been chosen per strategy in order to get more stable results.

**Population size** 100 individuals have been selected per generation, as this is the population size of the previous researches.

**Generations** 8 generations were achieved with the available time.

**Elitism** 10 of the best individuals per generation are carried to the next generation.

**Mutation rate** 5% this is the usual parameter used for mutation in most of the GA approaches.

**Tournament selection** 70% of the population per generation takes part of the generation. We have chosen a high value as this will create a greater chance that fitter individuals will be part of the genetic processes.

**Run time per experiment** 14 days

### Results and conclusions

As we already discussed, the results will be measured against the results of the previous curve steepness experiment.

### GA experiment

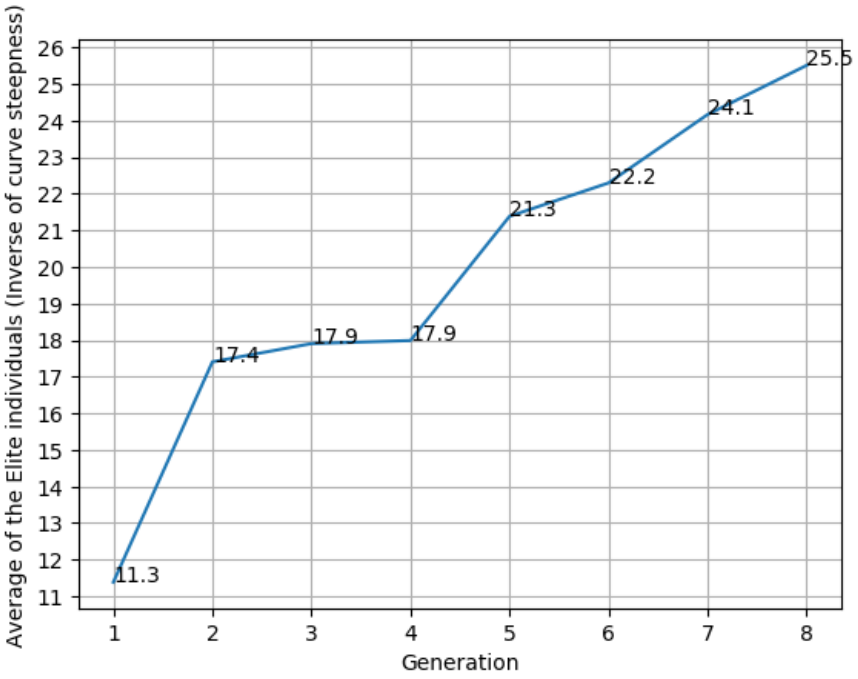


Figure 5.14: Mean values of the Elite strategies over the generations of Notepad curve steepness. (long experiment)

As we can see in the Figure 5.14 there is still a stable increase per generation until the eight generation. Therefore, more generations will most likely result in better results.

### 1000 runs of length 100

A boxplot of the results is given in Figure 5.15. Please note that for showing the results we use the inverse of the fitness function.

The results of the Kruskal-Wallis H-test is $0.0043269281141251 84 < 0.05$ which means that we reject the null hypothesis. The mean values are given in Table 5.10. We can see that the results of the long experiment are better than the ones achieved by the shorter one through both the boxplot and table of the mean values. Thus, we can conclude that introducing more evaluations per strategy does indeed give us better and more stable results during the GA experiment.

| | Mean (average) |
|---|---|
| First GA strategy | 25.7 |
| Long experiment GA strategy GA strategy | 26.5 |

Table 5.10: Table of the mean values of the results of Notepad 1000 runs of length 100. (long experiment)

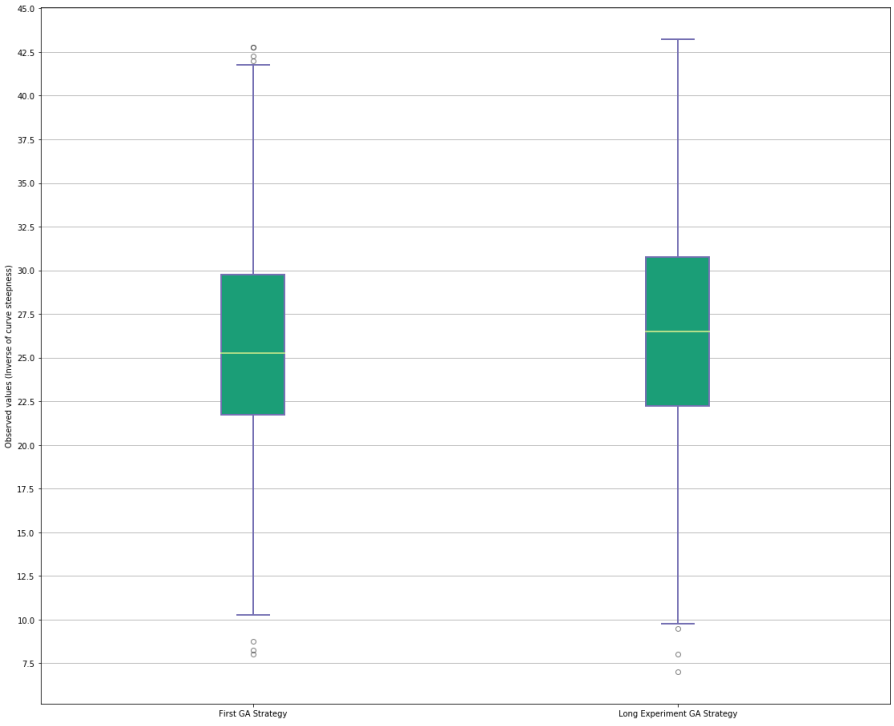


Figure 5.15: Boxplot of the Notepad 1000 runs of length 100. (long experiment)

### 30 runs of length 500

A boxplot of the results is given in Figure 5.16. Please note that for showing the results we use the inverse of the fitness function.

The results of the Kruskal-Wallis H-test is $0.5893839272875354 > 0.05$ which means that we can not reject the null hypothesis. The mean values are given in Table 5.11. In this experiment the results between the new strategy and the old one are not much different, but we can see less variance in the new one. Thus, we can still conclude that it achieved slightly better results.

| | Mean (average) |
|---|---|
| First GA strategy | 70.1 |
| Long experiment GA strategy GA strategy | 70.9 |

Table 5.11: Table of the mean values of the results of Notepad 30 runs of length 500. (long experiment)

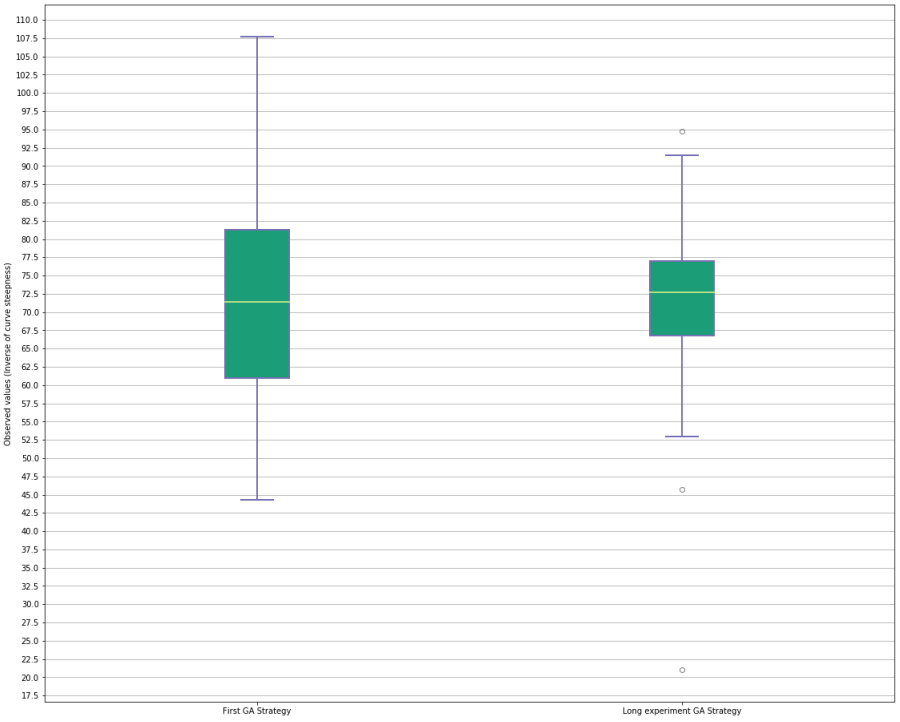


Figure 5.16: Boxplot of the Notepad 30 runs of length 500. (long experiment)

### 5.4.2 Unique states long experiment

This experiment was performed on the unique states fitness function introduced by Theuws [37]. It was able to achieve twelve generations until the time to stop it. In order to evaluate the strategy from this experiment, we are going to compare it to the RAS.

**GA Set-up**

**SUT** Notepad (The notepad version is related to the Windows version given in Appendix B)

**Number of actions per strategy** 100 actions have been chosen to be performed per evaluations. This is chosen due to being used in the previous research and due to time constraints.

**Evaluations per strategy** 15 evaluations have been chosen per strategy in order to get more stable results.

**Population size** 100 individuals have been selected per generation, as this is the population size of the previous researches.

**Generations** 13 generations were achieved with the available time.

**Elitism** 10 of the best individuals per generation are carried to the next generation.

**Mutation rate** 5% this is the usual parameter used for mutation in most of the GA approaches.

**Tournament selection** 70% of the population per generation takes part of the generation. We have chosen a high value as this will create a greater chance that fitter individuals will be part of the genetic processes.

**Run time per experiment** 14 days

### Results and conclusions

The evaluations are going to be performed similarly to the curve steepness experiment. The best GA strategy from the GA experiment is going to be compared to the RAS.

### GA experiment

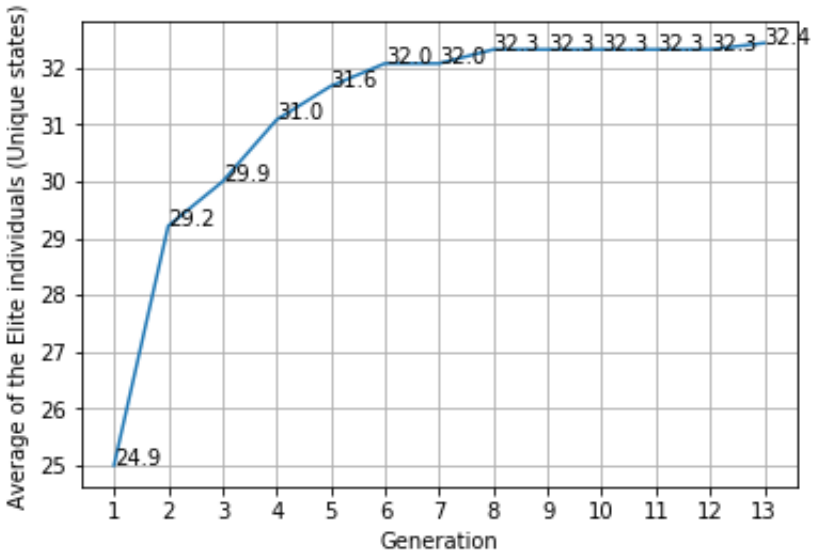


Figure 5.17: Mean values of the Elite strategies over the generations of Notepad unique states. (long experiment)

As we can see from the graph in Figure 5.17 there is no significant improvement with the generations after the eight one. This is different than the results of the other GA experiments and could be caused by the fitness function.

### 1000 runs of length 100

A boxplot of the results is given in Figure 5.18. Please note that for showing the results we use the inverse of the fitness function.

The results of the Kruskal-Wallis H-test is $3.5471035644321206e-127 < 0.05$ which means that we reject the null hypothesis. The mean values are given in Table 5.12. As we have rejected the null hypothesis and we can see that the results of GA strategy have higher values than the ones from RAS, we can conclude that the GA strategy outperformed the RAS.

| | Mean (average) |
|---|---|
| GA strategy | 28 |
| RAS | 20.2 |

Table 5.12: Table of the mean values of the results of Notepad 1000 runs of length 100. (long experiment unique states)

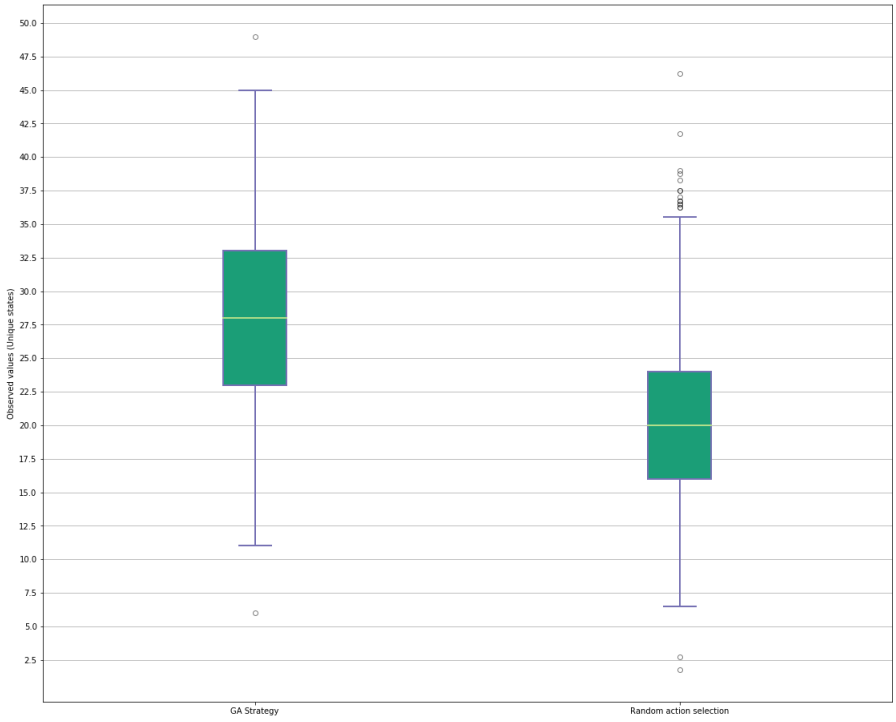


Figure 5.18: Boxplot of the Notepad 1000 runs of length 100. (long experiment unique states)

### 30 runs of length 500

A boxplot of the results is given in Figure 5.19. Please note that for showing the results we use the inverse of the fitness function.

The results of the Kruskal-Wallis H-test is $0.2034266422872618 > 0.05$ which means that we can not reject the null hypothesis. The mean values are given in Table 5.13. Although we can not reject the null hypothesis, we can still see that the results of the GA strategy seem to be a bit better than the RAS as the mean value of the results is larger.

| | Mean (average) |
|---|---|
| GA strategy | 59.6 |
| RAS | 57 |

Table 5.13: Table of the mean values of the results of Notepad 500 runs of length 30. (long experiment unique states)

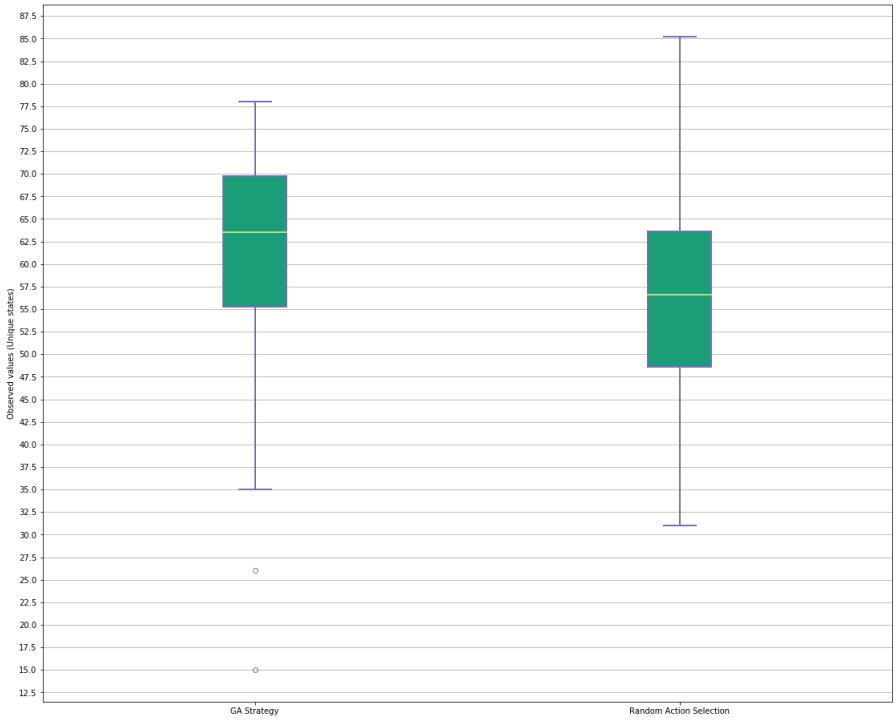


Figure 5.19: Boxplot of the Notepad 30 runs of length 500. (long experiment unique states)

# Chapter 6

# Conclusions and future work

The research of this thesis was conducted as part of the IVVES project. It focused on using the scriptless GUI testing tool TESTAR together with the AI approach GA in order to produce good testing results. We already discussed in Chapter 1 that GUI tests used to be executed by humans and moved towards a semi-automated approach which recorded the executed tests or wrote them as scripts so that they can be used at a later point in time. This tends to be costly as those tests need to be maintained and new ones should be produced in order to keep up with the evolution of the related GUIs. An alternative approach is using a completely automated approach which lowers the expenses. However, these approaches are exhibiting random behaviour and rarely satisfy the desired testing criteria. In order to improve this type of testing, the current thesis focused on using the GA strategy approach instead of the random one. Previous experiment had already been conducted, but the approach still required further research.

One of the directions described as future work [37] was looking into different evaluation criteria. Another topic which seemed important to discuss is whether the approach in itself is indeed better than the random one. This raised the following research question:

**RQ:** Can the strategy-based GA approach improve the effectiveness and performance of the existing automated GUI-based testing tool?

As we discussed in Chapter 1 we divided the question in the following sub-questions:

**RQ1.1:** What kind of evaluation criteria can be used in the strategy-based GA approach?

**RQ1.2:** Will the GA strategy approach be effective while using different evaluation criteria?

**RQ1.3:** Is selecting a strategy through GA better than selecting it through random selection?

In order to answer the first sub-question we performed research on the related work in the field in Chapter 3 and created three new fitness functions which are described in Chapter 4.

After performing the research we concluded that the general trend in GA-based GUI testing seems to be using an event-flow graph together with specific sequences as individuals. However, we already argued in Chapter 5 that the strategy approach is superior as it will in the end result in a higher number of test sequences and it will not become obsolete after modification of the GUI. In order to further strengthen this claim, the thesis aimed to compare the results of the second experiment to prior work [30]. However, the difference in the set-up of this thesis and the prior work is quite different as discussed in Section 5.2. Future work should focus on using specific sequences together with the TESTAR state model in order to determine whether the strategy approach or the sequence approach is superior.

We performed separate experiments on each of the fitness functions. On two of them, the curve steepness and abstract states, with GA we were able to achieve results which in seven out of eight cases would outperform the random action selection mechanism. This is described in the evaluation of the experiments in Sections 5.1 and 5.2. Thus, we can conclude that those strategies

were able to focus towards the specific fitness functions. However, in the third experiment which aimed to find specific bugs, we failed to replicate the known bugs. One of the bugs was detected, but not often enough to consider it a success, while the other was not found at all. As the bugs were quite specific, future work can determine whether the strategy approach is still too abstract in order to find them. Nonetheless, from the first two experiments we can see that if the goal is not too specific the strategy approach satisfies the evaluation criteria better than the random approach.

In order to answer the second sub-question, each of the experiments was performing evaluation with the random action selection mechanism (RAS) together with the strategies. In the curve steepness experiment we could clearly see that the GA approach outperformed the RAS on all of the evaluations. In the state model experiment, the GA approach still had a better performance on three of the four evaluations and a similar one on the fourth evaluation. This is most likely due to the lower number of evaluations during the GA experiment, which affects how good the GA strategy will perform, while it has no effect on the RAS. The third experiment came as a surprise, as the RAS outperformed the strategy by finding the bug more often. We already discussed during the experiment in Section 5.3, that finding the bug by both the RAS and the strategy is most likely due to luck and we should not use that experiment to make conclusions. The fourth experiment also showed that the GA strategy approach outperforms the RAS.

As for the third sub-question due to time constraints we were able to evaluate a strategy from a random set only on the curve steepness experiment. It was abbreviated as Best Random Strategy (BRS). The GA strategies outperformed the BRS on all of the evaluations. Thus, we can conclude that the GA approach is indeed successful in improving the strategies.

To answer the question:

**RQ:** Can the strategy-based GA approach improve the effectiveness and performance of the existing automated GUI-based testing tool?

The GA strategy approach seems to be effective in exploring the GUI, even if the evaluation criteria is more concrete, such as the curve steepness. This is shown, by all of the evaluations, apart from the bug finding, as there was at least one GA strategy that was performing on par or better than the RAS. However, it was ineffective in finding the already existing bug reports that we discussed in Section 5.3.

Although the experiments achieved favourable results, more thorough experiments should be performed. As we often see, both the number of evaluations and generations would improve the results of the GA experiments. Following the results of the experiments, we can propose the possible directions in which the experiments can be extended:

## Future work

**TESTAR** TESTAR needs to be investigated further, as it sometime causes unpredictable behaviour. Most of the problems were solved during this research, however a specific problem is the filtering of actions. For Notepad, even though the open file button was filtered, the action would still be performed in some cases. In VLC, some of the open file buttons (VLC has a high number of them), would could not be filtered out. This is most likely to the incorrect implementation of the accessability API in VLC, which has been mentioned as a problem in the research by Theuws. The bug caused a couple of evaluations to go wrong, as it would delete crucial files to the experiment.

**ECJ** Genetic programming tends to have a high number of parameters related to it. It also has different selection processes and rarely can someone conclude whether one is better than the other. The ones used in this thesis are Elitism and Tournament selection, however further research should focus on finding better parameters for them or exploring different alternatives. Up to date, we were unable to find suggestions about the perfect values of the parameters.

This is expected, as most likely they tend to differ per type of experiment.

**Statistic tests** In the current thesis we made the assumptions that by performing the Kruskal-Wallis H-Test we would be certain that the results in different groups are different, and thus conclude that one is better than the other based on the values. However, there were cases where even though the test was not distinguishing between the different groups, some of them had significantly higher mean values or medians than the other. Thus, more research is required in order to solidify the claim that one group is better than the other in those cases.

**More experiments** The time constraints are an issue due to the long run time of the experiments, especially if the state model is involved. As we have already seen, the number of evaluations is quite important in correctly ranking the strategies during the GA experiments. The exact number for stabilizing the evaluations is not known, but we would suggest that further research should look into that and use it during the GA experiments. However, that may prove difficult as it will most likely depend on both the fitness function and the number of actions per evaluation. The GA experiments also tend to keep improving with the generations and only 10 of them are not nearly enough to achieve the best results. The evaluations on different SUTs also have differing results and thus more SUTs should be included in the research. Most likely increasing the number of SUTs will always give us more information, but perhaps one should aim for at least 5 as most of the related work in the field that was performing on multiple SUTs was aiming for at least 5.

**Sequence vs Strategy** It would be interesting to fully compare the strategy approach and the sequence approach. As there is no research conducted in a way that it would be directly comparable to the results of this thesis, it is hard to give strong conclusions about it. However, if a research is conducted on TESTAR and the state model with using the sequence approach it would be easy to compare to the strategy approach.

**Fitness function** Finally our suggestion of the fitness function would be to combine all three of the fitness functions used in this thesis. The first one seems to be beneficial for the industry as it would provide shorter sequences of better quality. However, it currently uses the unique states information directly from TESTAR. The state model of TESTAR has been developed in order to determine those unique states in a better and perhaps more correct way. It also provides information about the visited states, as well as an estimated value on the percentage of states that were covered. Thus, one should aim to calculate the first fitness function, but replace the number unique states with the number of abstract states from the model. This has the downside that it will drastically increase the run time of the experiment due to using the state model. Finally, it would be valuable to add bug finding to the fitness function. Even though the current experiment was not successful in this, if a strategy can indeed find a bug, it should have better fitness value than one which can not. The bug finding part of the fitness function such emphasize on how fast a bug is found - finding a bug in 5 actions should be better than finding a bug in 500.

# Bibliography


[1] Cyberduck bug report, https://trac.cyberduck.io/ticket/11075. 43

[2] Ecj implementation by theuws, https://github.com/guyt07/thesis/tree/feature/prepare-ecj-for-testar. 15

[3] Itea eu 3, https://itea3.org/l. 1

[4] Ivves, https://ivves.weebly.com/. 1

[5] Orient db, https://www.orientdb.org/. 23

[6] Paint.net bug, https://tweakers.net/downloads/50696/paint-punt-net-429.html. 43

[7] T. E. J. Vos A. I. Esparcia-Alcazar, F. Almenar and U. Rueda. Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. Master's thesis, 2018. 12, 16

[8] Bestoun Ahmed, Mouayad Sahib, and Moayad Potrus. Generating combinatorial test cases using simplified swarm optimization (sso) algorithm for automated gui functional testing. *Engineering Science and Technology, an International Journal*, 17, 08 2014. 18, 19, 24

[9] Kolströmc P Alégroth E, Feldt R. Maintanenance of automated test suites in industry: An empirical study on Visual GUI Testing. Master's thesis, 2016. 1, 16

[10] H. Bunke, M. Last, and A. Kandel. *Artificial Intelligence Methods in Software Testing.* EBSCO ebook academic collection. World Scientific, 2004. 1, 16

[11] Chang Wook Ahn and R. S. Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 7(4):367–385, 2003. 9

[12] M. de Groot. Smarter Monkeys: Using evolutionary computing to improve black box monkey testing on a Graphical User Interface. Master's thesis, 2018. 12, 16, 17

[13] Cse Dept and N. Malmurugan. Automated gui test cases generation with optimization algorithm using a model driven approach. 2013. 19

[14] Fathi Essalmi and Leila Jemni Ben Ayed. Graphical uml view from extended backus-naur form grammars. In *Sixth IEEE International Conference on Advanced Learning Technologies (ICALT'06)*, pages 544–546. IEEE, 2006. 12

[15] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In *DATA*, pages 373–380, 2018. 7

[16] Ahmed Ghiduk, Mary Harrold, and Moheb Girgis. Using genetic algorithms to aid test-data generation for data-flow coverage. pages 41–48, 12 2007. 18

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery. 11

[18] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and design of selenium webdriver automation testing framework. *Procedia Computer Science*, 50:341–346, 2015. 3

[19] Andres Gonzalez and Loretta Guarino Reid. Platform-independent accessibility api: Accessible document object model. In *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, W4A '05, page 63–71, New York, NY, USA, 2005. Association for Computing Machinery. 3

[20] S. Huang, M. B. Cohen, and A. M. Memon. Repairing gui test suites using a genetic algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 245–254, 2010. 18

[21] Y. Huang and L. Lu. Apply ant colony to event-flow model for graphical user interface test case generation. *IET Software*, 6(1):50–60, 2012. 18

[22] G. Latiu, O. Cret, and L. Vacariu. Graphical user interface testing optimization for water monitoring applications. In *2013 19th International Conference on Control Systems and Computer Science*, pages 640–645, 2013. 18

[23] G.I. Latiu, Octavian Creţ, and Lucia Vacariu. Graphical user interface testing using evolutionary algorithms. pages 1–6, 01 2013. vi, 17

[24] G. I. Laţiu, O. Creţ, and L. Văcariu. Graphical user interface testing using evolutionary algorithms. In *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2013. 18

[25] L. Lu and Y. Huang. Automated gui test case generation. In *2012 International Conference on Computer Science and Service System*, pages 582–585, 2012. 18

[26] Thomas W MacFarland and Jan M Yates. Kruskal–wallis h-test for oneway analysis of variance (anova) by ranks. In *Introduction to nonparametric statistics for the biological sciences using R*, pages 177–211. Springer, 2016. 28

[27] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995. 9

[28] M. Mitchell. *An Introduction to Genetic Algorithms*. A Bradford book. Bradford Books, 1998. 9

[29] Ad Mulders. Masters thesis, Open Universiteit. Master's thesis, 2020. vi, vi, 7, 8, 26

[30] A. Rauf, S. Anwar, M. A. Jaffer, and A. A. Shahid. Automated gui test coverage analysis using ga. In *2010 Seventh International Conference on Information Technology: New Generations*, pages 1057–1062, 2010. 18, 21, 25, 26, 36, 38, 41, 54

[31] U. Rueda, A. Esparcia-Alcázar, and Tanja E. J. Vos. Visualization of automated test results obtained by the testar tool. In *CIbSE*, 2016. 7

[32] R. Sagayam and K. Akilandeswari. Comparison of ant colony and bee colony optimization for spam host detection. 2012. 19

[33] Rolv Seehuus, Amund Tveit, and Ole Edsberg. Discovering biological motifs with genetic programming. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 401–408, New York, NY, USA, 2005. Association for Computing Machinery. 21

[34] M. Soffa and A. Memon. A comprehensive framework for testing graphical user interfaces. 2001. 18

[35] Dr. Praveen Srivastava and Tai-Hoon Kim. Application of genetic algorithm in software testing. *International Journal of Software Engineering and Its Applications*, 3, 11 2009. 18, 19

[36] Nelly Condori-Fernández Sebastian Bauersfeld Tanja E.J. Vos, Peter M. Kruse and Joachim Wegener. Testar: Tool support for test automation at the user interface level. 2015. vi, 3, 4

[37] G. Theuws. Masters thesis, Open Universiteit. Master's thesis, 2020. vi, viii, viii, 2, 10, 11, 13, 14, 15, 16, 20, 23, 26, 27, 50, 54

# Appendix A

# System information

| Item | Value |
|---|---|
| OS Name | Microsoft Windows 10 Pro |
| Version | 10.0.18363 Build 18363 |
| Other OS Description | Not Available |
| OS Manufacturer | Microsoft Corporation |
| System Name | DESKTOP-SUVG3P3 |
| System Manufacturer | VMware, Inc. |
| System Model | VMware7,1 |
| System Type | x64-based PC |
| System SKU | Unsupported |
| Processor | Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz, 2793 Mhz, 1 Core(s), 1 Logical Processor(s) |
| Processor | Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz, 2793 Mhz, 1 Core(s), 1 Logical Processor(s) |
| BIOS Version/Date | VMware, Inc. VMW71.00V.7581552.B64.1801142334, 14/01/2018 |
| SMBIOS Version | 2.7 |
| BIOS Mode | UEFI |
| BaseBoard Manufacturer | Intel Corporation |
| BaseBoard Product | 440BX Desktop Reference Platform |
| BaseBoard Version | None |
| Platform Role | Desktop |
| Secure Boot State | Off |
| PCR7 Configuration | Binding Not Possible |
| Windows Directory | C:\Windows |
| System Directory | C:\Windows\system32 |
| Boot Device | \Device\HarddiskVolume2 |
| Locale | United Kingdom |
| Hardware Abstraction Layer | Version = "10.0.18362.752" |
| Username | Not Available |
| Time Zone | Romance Summer Time |
| Installed Physical Memory (RAM) | 8.00 GB |
| Total Physical Memory | 8.00 GB |
| Available Physical Memory | 2.98 GB |
| Total Virtual Memory | 15.0 GB |
| Available Virtual Memory | 2.83 GB |
| Page File Space | 7.00 GB |
| Page File | C:\pagefile.sys |
| Kernel DMA Protection | Off |
| Virtualisation-based security | Not enabled |

Figure A.1: System information about the remote desktop on which the experiments were performed.

# Appendix B

# Strategies found during the experiments

## B.1 Curve steepness Notepad strategies

**BRS**

```
if (drag-actions-available):
    then (random-action)
else
    (random-unexecuted-action-of-type click-action)
```

**First GA strategy**

```
if ((number-of-actions-of-type random-unexecuted-action) ==
(random-action-of-type-other-than random-action-of-type-other-than
                                  random-most-executed action)):
then
    if not (left-clicks-available):
        then previous-action
    else :
            (random-unexecuted-action-of-type random-action)
else:
    (random-action-of-type click-action)
```

**Second GA strategy**

```
    (random-action-of-type-other-than type-action)
```

**Third GA strategy**

```
(random-unexecuted-action-of-type type-action)
```

## B.2 Curve steepness VLC strategies

**BRS**

```
(random-unexecuted-action-of-type hit-key-action)
```

**First GA strategy**

```
random-action-of-type-other-than : (different than the ones below)
    if ((number-of-actions-of-type drag-action) >
    (number-of-actions-available-of-type random-unexecuted-action)):
    then
        (random-unexecuted-action-of-type
            random-least-executed-action)
    else:
        (random-action-of-type-other-than click-action)
```

**Second GA strategy**

```
(random-action-of-type-other-than hit-key-action)
```

**Third GA strategy**

```
(random-action-of-type-other-than
    random-action-of-type-other-than
            random-action-of-type-other-than drag-action)
```

## B.3 State model Notepad strategies

**First GA strategy**

```
if not (random-type == random-type):
    then
        (random-unexecuted-action)
else:
    (random-action-of-type-other-than hit-key-action)
```

**Second GA strategy**

```
if (number-of-action == number-of-unexecuted type-actions):
then
    (random-most-executed-action)
else:
    (random-unexecuted-action)
```

**Third GA strategy**

```
    (random-unexecuted-action-of-type hit-key-action)
```

## B.4 State model VLC strategies

**First GA strategy**

```
random-unexecuted-action-of-type: (all below)
     if (type-action-available and
             drag-action-available and
                 left-click-available):
     then
         (previous-action)
     else:
         (random-most-executed-action)
```

**Second GA strategy**

```
    (random-unexecuted-action-of-type click-action)
```

## B.5 Bug finding Cyberduck strategy

**GA strategy**

```
if ((number-of-previous-actions == random-number) and
        (number-of-unexecuted-drag-actions == number-of-left-clicks):
then
    (random-action-of-type-other-than  hit-key-action)
else if (type-action-available):
    then (previous-action)
else (random-unexecuted-action)
```

## B.6 Long experiment curve steepness

**GA strategy**

```
if (number-of-actions == number-of-unexecuted-left-clicks):
then
    (random-unexecuted-action-of-type: click-action)
else (random-unexecuted-action-of-type: click-action)
```

## B.7 Long experiment curve steepness unique states

**GA strategy**

```
random-unexecuted-action-of-type: (all below)
     if (left-clicks-available):
     then
         (random-most-executed-action)
     else:
         (previous-action)
```