Marion de Groot
851698512
April 2, 2018

Open Universiteit
www.ou.nl

Smarter Monkeys

Using evolutionary computing to improve black box monkey testing on a Graphical User Interface

Marion de Groot, 851698512

Smarter Monkeys

Using evolutionary computing to improve black box monkey testing on a Graphical User Interface

Date: April 2, 2018

Open University of the Netherlands, Faculty of Management,
Science and Technology
Master's Programme in Software Engineering

Graduation committee
Chair and secondary supervisor: Prof. Dr. Tanja E.J. Vos
Primary supervisor: Dr. Ir. Jeroen Keiren
Course code: IM9906

Contents

Summary

When delivering high quality software, testing is an important element of its development. Testing software can be time consuming, therefore automation of the process is practical and commercially interesting. Software can be tested from code level to system level. On the system level, one approach is to use the graphical user interface (GUI) to test specific scenarios. These scenarios require maintenance. Another approach is to randomly perform actions on the GUI, known as monkey testing. Several monkey testing tools exist, and one of them is TESTAR.

In monkey testing a key step is selecting the next action to perform, and the default strategy is to select a random action. Previous work on TESTAR [17] has attempted to find a better action selection strategy using evolutionary computing in a preliminary setup. In evolutionary computing, solutions to a problem are evolved using mutation and selection. In the previous work each individual in the evolution consists of a tree that represents an action selection strategy. This research builds on that work, expanding the evolution setup with more components and executing more experiments. ECJ [12] is used as the evolutionary computing engine. Three systems under test (SUTs) are used, on each of which four evolutions are run with ten generations of 100 individuals. To evaluate the fitness of the individual, the strategy it represents is used by TESTAR to run tests, and the metrics of the test are returned to ECJ. Each strategy is run 20 times with a sequence length of 100 actions.

From each of these evolution runs the three strategies with the best fitness value are selected. The performance of these strategies, together with the random action selection strategy and the strategy found in the previous work, are compared by running all of them on each of the three SUTs, with sequence lengths of 1000 and 100, 20 times each. The metrics of all runs are gathered to run statistical analyses on the performance of the strategies.

The evolution runs have not resulted in better action selection strategies than the random strategy or than the strategy of the previous research. However, major insight is gained into how evolutionary computing can help find better action selection strategies. A research agenda is proposed for further research on using evolutionary computing and machine learning to find better strategies.

Part I

Research context

Chapter 1

Introduction

In a world of mobile apps for every itch, an endless number of web sites and the upcoming Internet of Things, software is more ubiquitous today than ever before. Virtually everyone is a software user, one way or another. The average user tends to take for granted that the software works and forgets the effort that goes into developing it. If the software does not work as desired, the user easily diverts to one of many alternatives. The success of the software is thus closely related to its quality. Due to the omnipresence of software, strong competition of alternatives and high expectations of users, software quality is more important now than before. This quality comprises the features that are included and the ease of use, but also the absence of unexpected behavior like bugs and crashes.

One of the most used quality assurance techniques in industry is software testing. The complete absence of bugs is hard to guarantee, but extensive testing can certainly help find a fair share. As a rule of thumb it is assumed that more testing is better. Manually trying all possible combinations of paths to find all bugs within a reasonable amount of time is impossible for any system with more than a handful of options. It requires clicking the same buttons over and over again to see how they work in different situations or after changes have been applied, which is not only boring but also error-prone. Many software companies do not prioritize on testing and do not plan enough time for it in the project. Instead they use this time and money for adding functionality. Therefore test automation adds significant value for any company involved in software development.

Test automation can already be done on several levels. A diligent developer writes unit tests for all written code, so that when changes are made to the code, it is easy to see if all tests still pass. In Test Driven Development these tests are even written before the actual code [9]. JUnit [24] is a frequently used framework for Java to help make and execute unit tests. While unit tests verify that the smallest testable pieces of code work as intended, a framework like FitNesse [19] verifies that the code meets the specifications by testing entire modules. Both approaches test the code directly and both require manual creation of test cases and maintenance as the source code changes.

Not only the separate components of the back-end code but also their mutual communication should be tested, preferably automated. Integration tests verify that the different components of a system work together, while system tests test the system as a whole. One type of system testing is Graphical User Interface

(GUI) testing. Writing test code to interact with other code is easy, but it is more complicated to make test software interact with the GUI, which is made for humans. Automating GUI testing is therefore a challenge. A lot of effort has been put into addressing this challenge of automating the GUI testing process [7]. Several tools exist with different interaction methods and areas of application, each with their own advantages and disadvantages.

Capture-and-replay testing tools record and replay interactions of a human tester [15, 33]. This is a mature approach and several capture-and-replay tools are available, for example Appium [6] for mobile applications and Selenium [38] for browser-based applications. Test cases are easy to create by recording sequences of actions. When the cases are replayed the testing tool will notice when the system does not respond as expected. Capture-and-replay testing tools are ideal for *regression testing*, which verifies that the software still performs the same after it is changed. A caveat, as with more types of tools, is that the scenarios that are tested depend on the creativity of the test case creator. Ideally all possibilities are tested, not just the predictable ones [46]. Another challenge is the maintenance needed on the test cases when the GUI is affected by the changes [22].

Visual GUI testing tools try to address this maintenance problem by using image recognition to detect active buttons by their appearance [8, 5]. The tools can automatically find and click buttons, and generate test scenarios. Examples of visual GUI test tools are EyeAutomate [18] and EggPlant [13]. Less need for maintenance of test scenarios is an advantage, but one of the challenges of this approach is that not all controls are visually recognized as such [4, 3].

The problems of visual recognition of actions and the maintenance of test scenarios are both addressed by *traversal-based testing* tools. These tools scrape the GUI for possible actions using *reflection*, in which a plugin or API is used to recognize controls [2]. The detected actions are used to *traverse* the GUI. Whether traversal-based testing can be implemented depends on the availability of these plugins or APIs (Application Programming Interface), which can be application- or platform-specific.

The traversal-based testing tools use an approach called *monkey testing*. This means that the tool has no knowledge of the purpose of the actions. Based on the detected controls, actions are automatically selected and executed, most often at random. The tools approach the system under test (SUT) as a *black box*, which means that the tools do not need information on what happens within the application or access to the source code. There are no test cases to create or maintain, therefore preparation time is minimal, and so is maintenance for when the SUT has changed. A challenge is the test coverage of the SUT by these tools, as random action selection may result in less than optimal test scenarios.

An example of a traversal-based testing tool is GUITAR [21]. GUITAR uses a model that specifies the intended software behavior to generate test cases. When executing the test cases it checks the actual output with the specified output. A second example is a framework for monkey GUI testing introduced by Wetzlmaier [45], which makes use of so-called *guides*. A guide can be used to perform a certain action in a certain state that is needed to get to the next state, for example entering a username and password. The tool has been applied to test nightly builds of a software program, to detect if something is broken by the changes included in the build [45].

A third example of a traversal-based testing tool is TESTAR [37], a black box monkey testing tool that has been developed at the Polytechnical University of Valencia in Spain and the Open University of the Netherlands (in the context of the TESTOMAT project [39]). It uses the *Accessibility API*, which is also used by visually impaired people, to determine all available controls and their properties on the screen. Common problems like crashes are detected. Furthermore the tool can be customized for an application using for example oracles and action filters, to detect specific errors or error messages, or to avoid certain actions.

One of the key elements of TESTAR and similar traversal-based applications is action selection, determining the next action that will be performed on the SUT to traverse the GUI. Currently the default action selection strategy in TESTAR is random action selection, meaning that a random action is picked and executed from the set of actions that are available. As this strategy leaves the test scenario to chance, bad luck could result in limited coverage of the SUT. However, research has shown that random action selection can compete with manual test scenarios on performance, while requiring little preparation time [42].

Previous work on TESTAR [17] has attempted to improve over the random action selection strategy by using *evolutionary computing*, a computing method that draws inspiration from evolution as seen in nature. This method uses the principles of evolution as observed in nature, such as *mutations* and *natural selection*, to come to a solution that best fits the problem. The solutions are represented using a set of interchangeable components. A population is created, with possible solutions as individuals. Each individual is tested and the best performing ones are selected to create the next generation. They are mutated, recombining their components into new individuals. This way new solutions can be found [28, 1, 17].

A sub-area of evolutionary computing is genetic programming, in which the components are parts of a computer program. John Koza [25, 26] has expanded the work on genetic programming in the optimization of complex problems. Usually tree-based representations are used for the individuals, in which each node in the tree is an interchangeable component. Gross *et al.* [20] use genetic programming to evolve method call trees for object oriented unit tests. Wappler *et al.* [43] use genetic programming to generate unit tests.

In the work on TESTAR using evolutionary computing for action selection [17] the individuals are trees that represent action selection strategies, consisting of a basic set of components. The research setup is quite basic, using only one SUT for the evolution and a simple tree representation with a limited set of components for the individuals. Still the outcomes are promising, with the resulting strategy performing better than random action selection on some SUTs.

Because of those promising results, this research proceeds on the setup and findings of that research on TESTAR. The research setup is expanded using a larger set of components and more degrees of freedom in how to compose strategies from them. These changes lead to more possible strategies that can be generated and evolved. Because of this larger range of strategies the chance of well-performing strategies being among them is increased. Furthermore, multiple evolutions are run instead of one, to verify that the results of evolution are consistent. Three SUTs are used for the evolutions instead of one, to analyze if there is a difference between the results. All found strategies are run on all three SUTs to analyze if the best strategies are system-specific. All these tests lead to better insight into which

strategies are suitable for each SUT and how evolutionary computing can help find them.

To summarize, black box monkey testing can add significant value by providing a high quality, low maintenance test approach. The test monkeys now select actions randomly, while there might be a better strategy. Evolutionary computing can help in finding a better action selection strategy. Can the test monkeys be evolved to smarter monkeys? And can these smart monkeys help us solve the puzzle of perfect automated GUI testing? This research attempts to find the answers.

In Part I, Chapter 2 explains how TESTAR works and discusses previous research, while Chapter 3 explains the research approach, research questions and hypotheses. Part II elaborates on the setup of the research, explaining the composition of action selection strategies in Chapter 4, the setup for evolutionary computing in Chapter 5 and the architecture of the components added to the used software in Chapter 6. Part III presents the results, with the outcomes of the experiments analyzed and reflecting back on the hypotheses in Chapter 7, discussing the results in Chapter 8 and drawing conclusions in Chapter 9. A reflection on the research is given in Chapter 10.

Chapter 2

TESTAR

2.1 Introduction

TESTAR is a traversal-based black box monkey testing tool. Figure 2.1 shows the workflow of TESTAR when testing a system under test (SUT). When TESTAR starts the SUT, it uses the Accessibility API to scan the GUI for controls and their properties, which it interprets as the *state* of the SUT. After every scan, TESTAR evaluates what it has found, for example what actions can be executed and if there are error messages on the screen. Next, it selects one action from the possible actions on the detected controls. This selection is currently done randomly. It then executes this action on the SUT using the Accessibility API. After executing the action the GUI is scanned again to see if the state has changed. This loop is repeated until a specified condition is reached. Such a condition could be that an error is found or that the configured number of executed actions is reached.

TESTAR can be customized to specific applications by creating *oracles* and *action filters*. An oracle is a specification of a state that is either desirable or undesirable. By default oracles are included that detect system crashes or freezes. Using regular expressions, the tester can specify certain texts or text patterns, such as pop-ups that contain the text 'error'. More advanced oracles can be defined by adapting the default, Java-based protocol that is used to drive the tests. Action filters define which actions should not be executed, for example exiting the application.

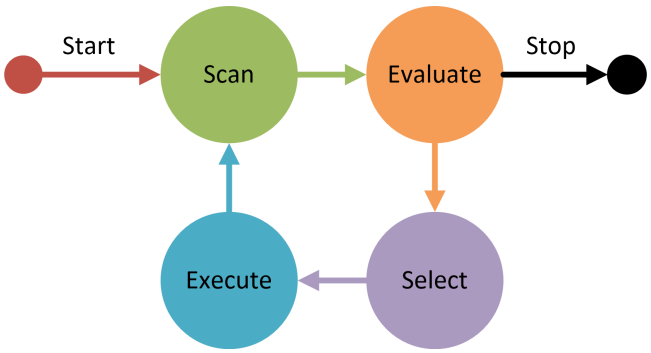

Figure 2.1: The cycle that TESTAR goes through when testing a SUT.

2.2 State transition model

The information that TESTAR gets from the Accessibility API includes which controls are available, their title, type (textbox, button, etc), status (enabled or disabled) and the possible actions. This set of controls and their properties at a given moment is interpreted by TESTAR as the state of the SUT. It is important to note that this state is solely based on the user interface and not on internal changes. When, after executing an action, a different set of controls or a difference in properties is detected, TESTAR concludes that the state has changed. TESTAR records this information as a transition to a new state. By performing action after action, TESTAR discovers states and transitions and composes a state transition model. An example of a state transition model is given in Figure 2.2.

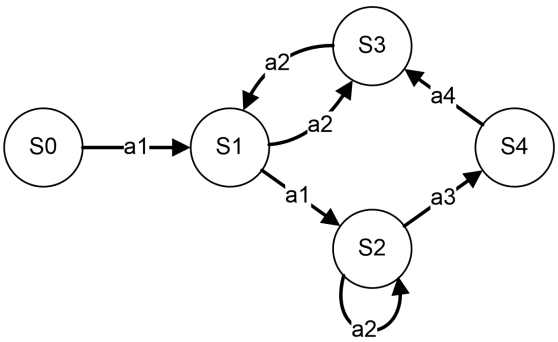

Figure 2.2: An example of a state transition model. The states are indicated with s0 to s4, and the transitions are labeled with the actions causing them, a1 to a4.

One state transition model is created for each test run, which is delimited by a predefined number of steps or ends when an error occurs. Models of several test runs can be compared, or can be aggregated into a single model to evaluate the total performance of multiple runs. A limitation is that the model only contains the states that have been reached and the actions that have been detected. It should not be interpreted as a complete model of the user interface and is therefore not a reliable basis for measuring test coverage.

2.3 Test metrics

Metrics are needed to measure how well TESTAR tests. A perfect test strategy would find all faults in a SUT within the shortest possible time. Because the number of faults is unknown, a tester never knows when to stop looking for more faults, or what percentage has been found. Instead, test tools usually measure what percentage of the total SUT has been covered by the tests.

Coverage of the tests is often measured by the source code coverage [31]. To measure code coverage, the source code needs to be available. Research conducted with TESTAR [29] looked into measuring code coverage dynamically, using JaCoCo [23], a code coverage measurement tool for Java. This metric can then be returned to TESTAR, to be included in its test results. As TESTAR is a black box testing tool, the source code or other indicators of the total size of the SUT are not necessarily available.

The state-transition-model of TESTAR explained above can be used to gain insight in how well the SUT has been tested, even though the model is incomplete. In the context of TESTAR several definitions of good testing have been proposed that can be measured [16, 36, 17]. These and other metrics that TESTAR distills from the state transition model are listed below.

Graph states the number of unique states in the state transition model

Graph actions the number of unique actions that have been executed, shown as transitions in the model

Test actions the total number of non-unique actions that have been executed

Abstract states the number of unique abstract states visited, in which an abstract state is a group of states that are the same when a certain set of properties is disregarded (for example the color of controls)

Longest path the length of the longest path in the state transition model

State coverage the minimum and maximum state coverage, defined as the lowest and highest rate of executed actions over the total number of available actions of a state

Even with an incomplete state-transition-model, these metrics can be used to compare test coverage of action selection strategies. When more graph states are visited, a larger part of the SUT is covered and there is a larger chance of faults being detected. The above mentioned works have used Graph states, Abstract states, Longest path and State coverage. Chapter 5 explains the metrics used in the presented work.

Apart from the state transition model, TESTAR also observes and reports other events that can be used as metrics or for diagnostics.

Memory usage the peak amount of memory used by the SUT

CPU usage the peak amount of CPU used by the SUT

Faults the number of faults detected by TESTAR, according to the configured oracles

The first two metrics do not indicate how well the SUT is tested, but they can be used to detect faults, for example memory leaks. The number of faults may seem to be the most important metric, as the goal of testing is to find faults. However, the number of faults in a SUT is unknown and the chances of finding many of them in a short time are small. While a typical short test may have covered 70 states, it may only have found one fault. The number of faults encountered therefore depends too much on chance to make it a reliable metric.

2.4 Previous work

Previous work has touched upon various aspects of TESTAR, of which an important one is the selection of the next action. The default action selection strategy of TESTAR is random action selection. Research comparing TESTAR using random

action selection with manual testing shows that the random strategy can compete with manually creating and running test cases on code coverage and fault finding, while requiring less preparation and maintenance [42].

In other previous work $\mathcal{Q}$-learning, a machine learning algorithm [44], has been used as an action selection strategy for TESTAR [16]. The algorithm assigns a value to each available action, which is high when the action has not been executed yet or when the values of the actions in the next state are high. By selecting the action with the highest assigned value, the algorithm leads TESTAR through the GUI. The algorithm yields some positive results, but not always, not on all SUTs and generally not (much) better than random action selection.

2.5 Previous work on evolutionary computing

The presented work builds on previous work on TESTAR using evolutionary computing to evolve action selection strategies [17]. This section elaborately explains the setup and results of this previous work, so that the starting point for the presented work is clear. Throughout this report, this previous research is labeled *EARV*, which are the first letters of the last names of the authors. The setup of EARV is shown in Table 2.1, cited from the research paper. A similar table is shown in Chapter 5 comparing the setup of the presented work with that of EARV.

In evolutionary computing individuals are randomly generated from components and then evolved into hopefully fitter individuals. This is done by evaluating each of the individuals to determine their fitness, selecting some of them and performing mutations on the selected individuals to create new individuals. The fitter individuals have a larger chance of being selected for mutations. The assumption is that the individuals can pass on their fitness to their children, so the selections and mutations should result in an even fitter next generation.

In EARV the individuals are trees that represent action selection strategies, as shown in Figure 2.3. As the tree consists of components from a predefined set, trees can be automatically generated and components can be interchanged. EARV uses a basic setup with a limited set of components. The root of the tree is always an if-

Table 2.1: Parameters used in EARV [17]

Feature	Value
Population size	20
Max tree size	20
Functions	Pick, PickAny, PickAnyUnexecuted, And, Or, LessThan, Equals, Not
Terminals	nActions, nTypeInto, nLeftClick, previousAction, Random, typeLeftClick, typeTypeInto, Any
Evolutionary operators	Mutation and Crossover
Evolutionary method	Steady state
Selection method	Tournament of size 5
Termination criterion	Generating more than 30 states

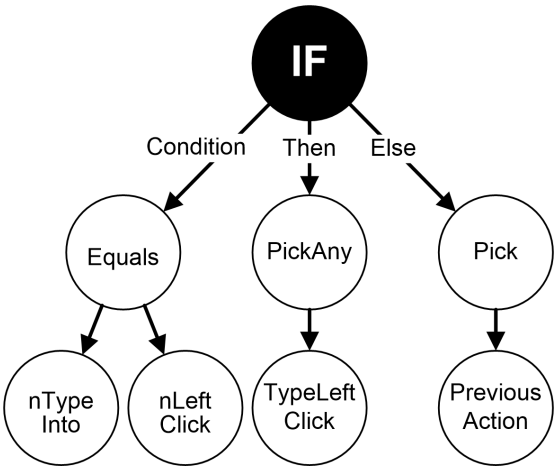

Figure 2.3: A tree-based representation of a strategy as used in EARV [17]

statement. The branches and leaves of the tree are booleans and action selections, with optional operators.

PonyGP [35] is used as the evolutionary computing engine. This is a simple python-based engine used for teaching purposes, to show the principles of evolutionary computing. Table 2.1 shows that a population of 20 individuals is used, with a maximum tree size (the number of nodes in a tree) of 20. The nodes are divided into functions and terminals. The evolutionary method used is the steady state model, which means that the population is not entirely changed at once, but partly, for example one individual at a time [14].

Tournament selection of size five is used. This means that five individuals are selected randomly from the total population, and of these five the fittest individual is picked for further processing. This processing is done using the evolutionary operators, in this case mutation (replacing part of the individual with a randomly generated part) and crossover (exchanging part of the individual with a part of another individual it is crossed with). The evolution is set to end as soon as a strategy is found that results in more than 30 visited states within the configured 100 actions.

The SUT used during the evolution is Microsoft PowerPoint. The strategy found using the evolution is then compared with the random strategy and the $\mathcal{Q}$-learning strategy described in Section 2.1. All three strategies are run on PowerPoint, Odoo (an open source Enterprise Resource Planning system) [34] and Testona (a commercial application for test case design) [40]. The metrics that are used for comparison are the Abstract states, the Longest path and the minimum and maximum state coverage.

Table 2.2 shows the results of EARV, comparing the strategy found with evolutionary computing (labeled the *EARV strategy*) to the $\mathcal{Q}$-learning strategy and random action selection. The results show that the EARV strategy performed best on PowerPoint and Odoo, measured by the Abstract states and the Longest path. On Testona, however, the random strategy performs better on these two metrics, while the EARV strategy has the highest maximum and minimum state coverage.

Table 2.2: Test results using the EARV strategy, $\mathcal{Q}$-learning or random action selection (RND), sorted from best to least score on test coverage parameters [17]

PowerPoint	Best		Least
Abstract states	EARV	$\mathcal{Q}$-learning	RND
Longest path	EARV	$\mathcal{Q}$-learning	RND
Max state coverage	RND	$\mathcal{Q}$-learning	EARV
Min state coverage	$\mathcal{Q}$-learning	EARV	RND
Odoo	Best		Least
Abstract states	EARV	$\mathcal{Q}$-learning	RND
Longest path	EARV	$\mathcal{Q}$-learning	RND
Max state coverage	EARV	$\mathcal{Q}$-learning	RND
Min state coverage	RND	$\mathcal{Q}$-learning	EARV
Testona	Best		Least
Abstract states	RND	EARV	$\mathcal{Q}$-learning
Longest path	RND	EARV	$\mathcal{Q}$-learning
Max state coverage	EARV	$\mathcal{Q}$-learning	RND
Min state coverage	EARV	$\mathcal{Q}$-learning	RND

Although the setup of EARV is limited, with a single evolution run on one SUT and a limited set of components, the EARV strategy performs well compared to the random action selection strategy and $\mathcal{Q}$-learning in many of the cases. This is a good starting point for further research with a more elaborate setup. The presented work will use a larger set of components with more degrees of freedom to allow a larger range of strategies to be created. Four evolutions will be run on three SUTs to have a more solid basis for conclusions on whether evolutionary computing helps in finding better action selection strategies. The more elaborate research will lead to more insight into how better action selection strategies can be found using evolutionary computing. The approach for this research is elaborated in the next chapter.

Chapter 3

Approach

This research builds on the previous research [17] (labeled EARV) and aims to find better action selection strategies using evolutionary computing. The research questions that lie at the base of this research are listed and explained below, and hypotheses with the expected answers to the research questions are formulated. The method of the research is explained and the research steps are elaborated.

3.1 Research questions

Main research question. Is a better action selection strategy for TESTAR found, specifically for each system under test (SUT), by using evolutionary computing with a more complex tree representation of the strategies?

The main research question reflects that the research builds on EARV [17], and expands it by using a more complex tree representation. More complex refers to more components and more degrees of freedom in the composition of these components into a tree. This will result in a larger range of possible strategies that can be generated, which can lead to a more refined strategy that fits the SUT better. A better action selection strategy refers to a strategy that results in a larger coverage of the SUT.

The following subquestions help in answering the main question.

Research question 1. Does the complex evolution setup result in a better strategy than random action selection?

As the aim of this research is to improve monkey testing, the first criterion for a better strategy is that it is better than random action selection. Based on the results of EARV it is expected that a strategy found with the complex evolution setup is better than random action selection.

Research question 2. Does the complex evolution setup result in a better strategy than the basic setup of EARV?

As this research builds on EARV with a more elaborate setup, the expectation is to get better results. The strategy found using the complex evolution setup is therefore compared to the EARV strategy. It is expected that the complex evolution setup yields better strategies than the basic setup.

Research question 3. Is there a difference in how strategies perform between SUTs?

If the evolution yields a good strategy, should this strategy be implemented in TESTAR to be used in every test? Or should the evolution be run again on every SUT, to find the best strategy specifically for that SUT? To answer these questions, it is important to know whether there is a difference in how a strategy performs between SUTs. It is expected that there is a difference, meaning that the definition of a good strategy is system-specific.

Research question 4. Does a strategy evolved on SUT A perform better on SUT A than on SUT B, and does it perform better on SUT A than a strategy evolved on SUT B?

If a strategy is indeed system-specific, it is expected that a strategy that results from the evolution using SUT A performs better on that SUT than on other SUTs. Likewise, it is expected that a strategy resulting from the evolution on SUT A performs better on SUT A than on SUT B.

The answers to these subquestions lead to an answer to the main question, with the expectation that the complex evolution setup leads to better action selection strategies specific for a SUT.

To summarize, the above research questions lead to the hypotheses below.

Hypothesis 1. A strategy found using the complex evolution setup is better than random action selection.

Hypothesis 2. A strategy found using the complex evolution setup is better than the EARV strategy.

Hypothesis 3. There is a difference between SUTs in how a strategy performs.

Hypothesis 4. A strategy evolved on SUT A works better on SUT A than strategies found on SUT B, and a strategy found on SUT A works better on SUT A than on SUT B.

With the final, main hypothesis:
Main Hypothesis. A better strategy is found specifically for each SUT by using the complex evolution setup.

3.2 Method

To find an answer to the research questions above, the research setup as shown in Figure 3.1 is used. Three SUTs are used, being the Windows 7 Calculator, VLC Media Player [41] and Testona [40]. In phase A four evolutions are run on each SUT, shown on the left side of the image. The evolutionary computing engine manages the population of individuals. Each individual contains a strategy. When an individual is evaluated, the strategy represented by the individual is sent to TESTAR. TESTAR then uses this strategy to select the next action in testing the SUT with a configured number of actions and returns metrics to the evolutionary computing engine. The metrics are used to determine the fitness of the individuals, which influences the evolution. At the end of the evolution the fittest strategy is stored in a list.

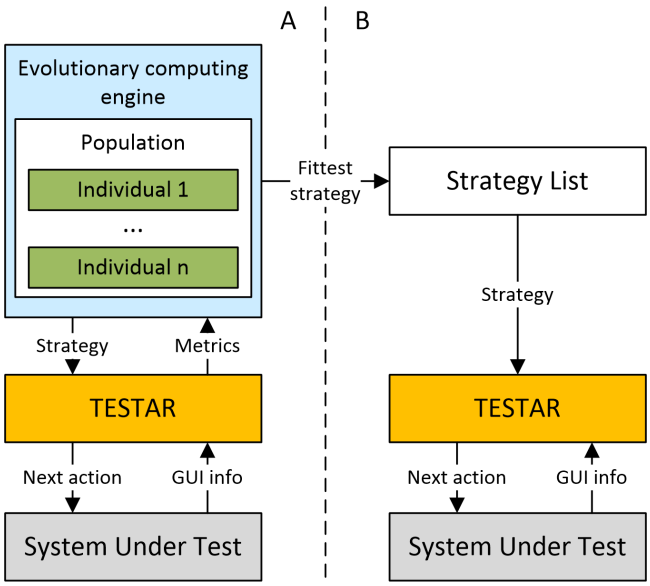

Figure 3.1: The setup of the research, with the evolution on the left and the tests for further analysis on the right.

Next in phase B, shown on the right of Figure 3.1, this list of strategies is used, together with the random strategy and the EARV strategy. All strategies on the list are run on all three SUTs, to be able to compare their performance across all SUTs. The test runs are longer than the ones in phase A, to be able to see the performance of the strategies in longer runs. The metrics of these tests are collected and analyzed. The statistical analyses Kruskal-Wallis and Mann-Whitney, as described in Wohlin *et al.* [47], are used to confirm whether the observed differences, if any, are significant.

Before these experiments can be executed, the setup as shown in Figure 3.1 is created, which is explained in Part II of this report. The components of the strategy trees are decided upon and the settings for the evolutionary computing engine are selected, as explained in Chapter 4 and 5. The evolutionary computing engine is connected to TESTAR, and TESTAR is prepared to be able to apply the strategies in its tests. The architecture of the changes to these two applications is described in Chapter 6.

Part III of this report describes the results of the experiments. The best strategies of each of the evolution runs in phase A of the experiments are listed. The performance of the strategies in phase B of the experiments is compared to the performance of the random strategy and the EARV strategy. The results are related back to the stated hypotheses, which are then accepted or rejected. The outcomes are discussed and used to propose a research agenda for further research on this topic. Conclusions are drawn and recommendations are given.

Part II

Research setup

Chapter 4

Building a strategy

4.1 Strategy trees

As introduced in Part I, a strategy in the context of this research is a rule according to which actions are selected to execute on the system under test (SUT) during black box monkey GUI testing. A simple strategy can be 'pick a random action' or 'pick a random click action'. In EARV [17] strategies always have an if-statement as the root, allowing the selected action to depend on a boolean value. A more complex rule is shown in Algorithm 4.1, using a nested if-statement.

Algorithm 4.1 An example of an action selection strategy

```
if there is a text field available then
    enter text
else
    if there is a button available then
        click a button
    else
        pick a random action
    end if
end if
```

To be able to automatically generate and evolve action selection strategies, the components of these strategies and how they fit together need to be decided. A tree-based structure is chosen for composing the strategies. The strategy of Algorithm 4.1 is represented as a tree structure in Figure 4.1. The tree-based approach is in line with EARV. Furthermore, the strategy tree is basically an abstract syntax tree, enabling a basic programming structure.

A tree consists of nodes, and nodes can have a number of children. The evolutionary computing engine will compose trees from a set of nodes, so this set of nodes needs to be defined. In Figure 4.1 the circles are the nodes of the tree, which are the if-statements, the booleans 'there is a text field' and 'there is a button', and the actions 'enter text', 'click button' and 'random action'.

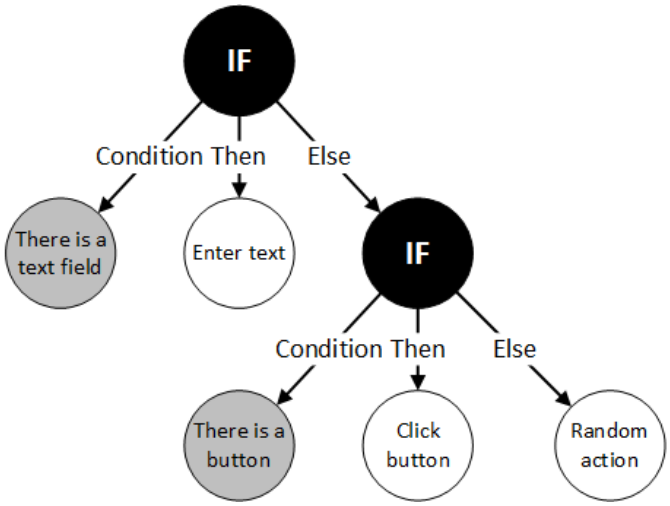

Figure 4.1: A strategy in the form of a tree.

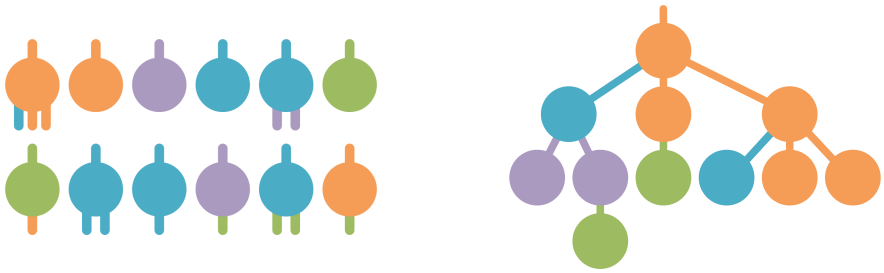

Figure 4.2: A symbolic representation of node types, with the colors representing types. The node types with child types are shown on the left. The right image shows an example of a tree generated from these node types.

4.2 Node and action types

Because not every node of a tree will fit anywhere to result in a valid strategy, nodes have different types. The evolutionary computing engine does not need to know what the nodes do, but it does need to know how the nodes can be combined. This is symbolically represented in Figure 4.2. The colors depict return types, and every node type has a return type, number of children and child types.

The return types encountered in Figure 4.1 are *actions* and *booleans*. The if-statement will return an action when evaluated, so the return type is also *action*. It has three children, of which the first one has to be a boolean, and the second and third have to be actions. Other types of nodes that can be used in a strategy are *actiontype* (for example type action or click action) and *number*. A node of type *actiontype* returns the type of an action, not the action itself. For example, the action type node 'click action' can be a child of the action node 'random action of type'. The return type *number* can be used by a boolean operator, for example 'equals'. The return type of the entire tree needs to be *action*, as the strategy is

Table 4.1: The Return types of the action selection components

Return type	**Explanation**	**Examples**
actiontype	Types of actions that TESTAR can do on the SUT	left click, right click, type
action	Actions that TESTAR can do on the SUT	left click action, type action, random action
number	A numeric value	Number of left click actions, Number of type fields, random number
boolean	A boolean value	True, False

used to select an action. All return types that are used in this research are listed in Table 4.1.

The *actiontype* return type can have a wide range of values. Table 4.2 lists all action types that TESTAR distinguishes. As the table shows, these action types are not distinct types, but a complex structure of simple and composite action types. For example, *MouseDown* and *MouseUp* are distinguished as two separate types, which together form the action type *Click*. When a mouse location is added using *MouseMove*, this gives a *ClickAt* action.

When TESTAR choses an action, the action can be of any of the listed types. The strategy components can use these action types to select an action of a certain type, for example 'random type action'. The action types marked in **bold** will be used in the strategy components in this research, while the underlined action types are used in EARV. EARV already included *Type* and *LeftClickAt* in components. For this research, components specifically referring to the types *HitKey* and *Drag* are included.

The action type *HitKey* includes the *Type* actions, but also hotkeys for quick access of functionality. The type *RightClickAt* is not specifically referred to in components, as it usually evokes a context menu, and clicking right consecutively usually does not result in new states. Other types are not used because they are too broad (*MouseAction*) or too specific (*LDoubleClickAt*) to be put in components. Note that actions of all types still have a chance of being selected, even if there is no specific component for the type.

Table 4.2: The action types that TESTAR distinguishes

Action type	**Parent action type(s)**
Action	
MouseAction	Action
KeyboardAction	Action
MouseMove	MouseAction
MouseDown	MouseAction
KeyDown	KeyboardAction
MouseUp	MouseAction
KeyUp	KeyboardAction
HitKey	KeyDown, KeyUp

Click	MouseDown, MouseUp
LeftClick	Click
RightClick	Click
DoubleClick	Click
LDoubleClick	LeftClick, DoubleClick
RDoubleClick	RightClick, DoubleClick
ClickAt	Click, MouseMove
LeftClickAt	ClickAt, LeftClick
RightClickAt	ClickAt, RightClick
DoubleClickAt	ClickAt, DoubleClick
LDoubleClickAt	DoubleClickAt, LeftClick
RDoubleClickAt	DoubleClickAt, RightClick
Type	HitKey
ClickTypeInto	ClickAt, Type
DropDown	Click, KeyDown
Drag	MouseDown, MouseUp, MouseMove
LeftDrag	Drag

4.3 Strategy nodes

Now that the return types and action types are decided on, a list of nodes can be created where the return type, the number of children and the return types of these children are defined. The list used for this research is shown in Table 4.3. The nodes are based on the list of nodes of EARV [17] with some additions. The added nodes are marked in **bold**. The if-statement was used in EARV as the fixed root of the tree. Here it is used as a node on itself, allowing nested if-statements and different nodes as the root.

Table 4.3: The action selection node types. The node types marked in **bold** are additions to the list of EARV [17].

Node type	Name	Return type	Child types
If (1) then (2) else (3)	if-then-else	action	(1) boolean (2) action (3) action
Random action	random-action	action	
The previously executed action	previous-action	action	
Random action of type (1)	random-action-of-type	action	(1) actiontype
Random unexecuted action	random-unexecuted-action	action	
Random unexecuted action of type (1)	random-unexecuted-action-of-type	action	(1) actiontype
Random action of type other than (1)	random-action-of-type-other-than	action	(1) actiontype
Random least executed action	random-least-executed-action	action	

Random most executed action	random-most-executed-action	action	
(1) greater than (2)	greater-than	boolean	(1) number (2) number
(1) equals (2)	equals	boolean	(1) number (2) number
(1) equals type (2)	equalstype	boolean	(1) actiontype (2) actiontype
(1) AND (2)	and	boolean	(1) boolean (2) boolean
(1) OR (2)	or	boolean	(1) boolean (2) boolean
NOT (1)	not	boolean	(1) boolean
There are type actions available	type-actions-available	boolean	
There are left clicks available	left-clicks-available	boolean	
There are drag actions available	drag-actions-available	boolean	
State has not changed after previous action	state-has-not-changed	boolean	
Click action	click-action	actiontype	
Type action	type-action	actiontype	
Drag action	drag-action	actiontype	
Hit key action	hit-key-action	actiontype	
Type of action of (1)	type-of-action-of	actiontype	(1) action
Random number	random-number	number	
Number of actions available	number-of-actions	number	
The number of previously executed actions	num-previous-actions	number	
The number of left click actions available	num-left-clicks	number	
Number of drag actions available	number-of-drag-actions	number	
Number of type actions available	number-of-type-actions	number	
Number of unexecuted type actions available	number-of-unexecuted-type-actions	number	
Number of unexecuted left click actions available	number-of-unexecuted-left-clicks	number	
Number of unexecuted drag actions available	number-of-unexecuted-drag-actions	number	
Number of available actions of type (1)	number-of-actions-of-type	number	(1) actiontype

Now that the components are defined, the evolutionary computing engine can be configured to create and evolve action selection strategies. The next chapter explains the setup of the evolutionary computing engine.

Chapter 5

Evolution setup

5.1 Introduction

Evolutionary computing, as introduced in Part I, draws inspiration from evolution as observed in nature in trying to solve complex problems. The individuals are possible solutions to the problem, which are then selected, mutated and evaluated. Several engines exist that facilitate evolutionary computing. EARV [17] uses PonyGP [35], which is a basic Python-based engine used for teaching purposes. For this research ECJ is chosen, a Java-based evolutionary computation research system [12]. This engine has more options than PonyGP, is more flexible and is built for research purposes.

Evolution in ECJ and similar engines is divided into several steps, as shown in Figure 5.1. Initially in the *creation* step a number of individuals is randomly generated from the components that are available, resulting in a *population*. Next each individual of the population is *evaluated* to determine its *fitness*, a measure for how good it is, or to what extent it *fits* the purpose. Then individuals are *selected* from the population with a selection algorithm that is usually based on both luck and fitness of the individuals. The next step is to perform *operations* on the selected individuals to create new individuals that form a new population, the next generation. The selection and operation steps are repeated until the desired population size is reached. The whole process is repeated for the desired number of generations.

Because in this research the individuals are action selection strategies, TESTAR is used in the evaluation step to determine the fitness of each individual, as shown in the sequence diagram in Figure 5.2. When an individual is evaluated, TESTAR is started with the strategy as a string parameter. TESTAR creates an *action selector* object from this string. This action selector has a method *getAction*, which selects and returns the next action to execute, based on the strategy. The action is then executed on the system under test (SUT), the GUI is scanned to analyze the new situation and the action selector is again asked for an action. This is repeated for the configured number of actions.

When the sequence length is reached, the SUT is closed, the metrics of the run are recorded and saved in a file and TESTAR exits. ECJ reads the metrics and stores them with the strategy. TESTAR is run again for the configured number of runs per individual. After that the fitness of the individual is determined based on the

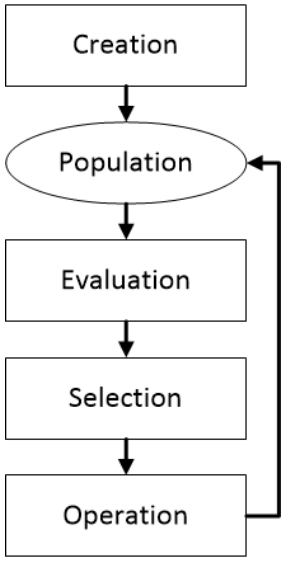

Figure 5.1: The steps in the evolution as implemented in ECJ.

metrics of all runs. Then the next individual is evaluated, until all individuals in the generation have a fitness value. ECJ then continues with the selection and operation steps. The whole process is repeated for the configured number of generations.

Evolution in ECJ can be customized using parameters. Running TESTAR for the fitness evaluation also requires the configuration of certain parameters. Section 5.2 describes the parameters that are considered relevant for this research. Section 5.3 aims to find the right settings for these parameters.

5.2 Parameters

5.2.1 General

ECJ supports several types of evolution setups. The default generational model is used, in which the entire population is replaced by its offspring after every generation. The number of generations can be configured. This is unlike the steady state model used in EARV [17], in which the population is continuously changed part by part.

Furthermore Koza-style evolution is used, as introduced in Chapter 2. The Koza-style evolution is based on tree-based individuals, which fits the research setup. The ideal fitness of an individual in Koza-style evolution is zero, while the worst fitness is infinity. If higher is better in the original value, the fitness can be standardized to zero for example by dividing one by the found value.

5.2.2 Creation

Initially ECJ will randomly generate a population of a configurable *population size* from the available node types (see Chapter 4). After the evaluation phase the selection and operation steps are repeated until the same population size is again

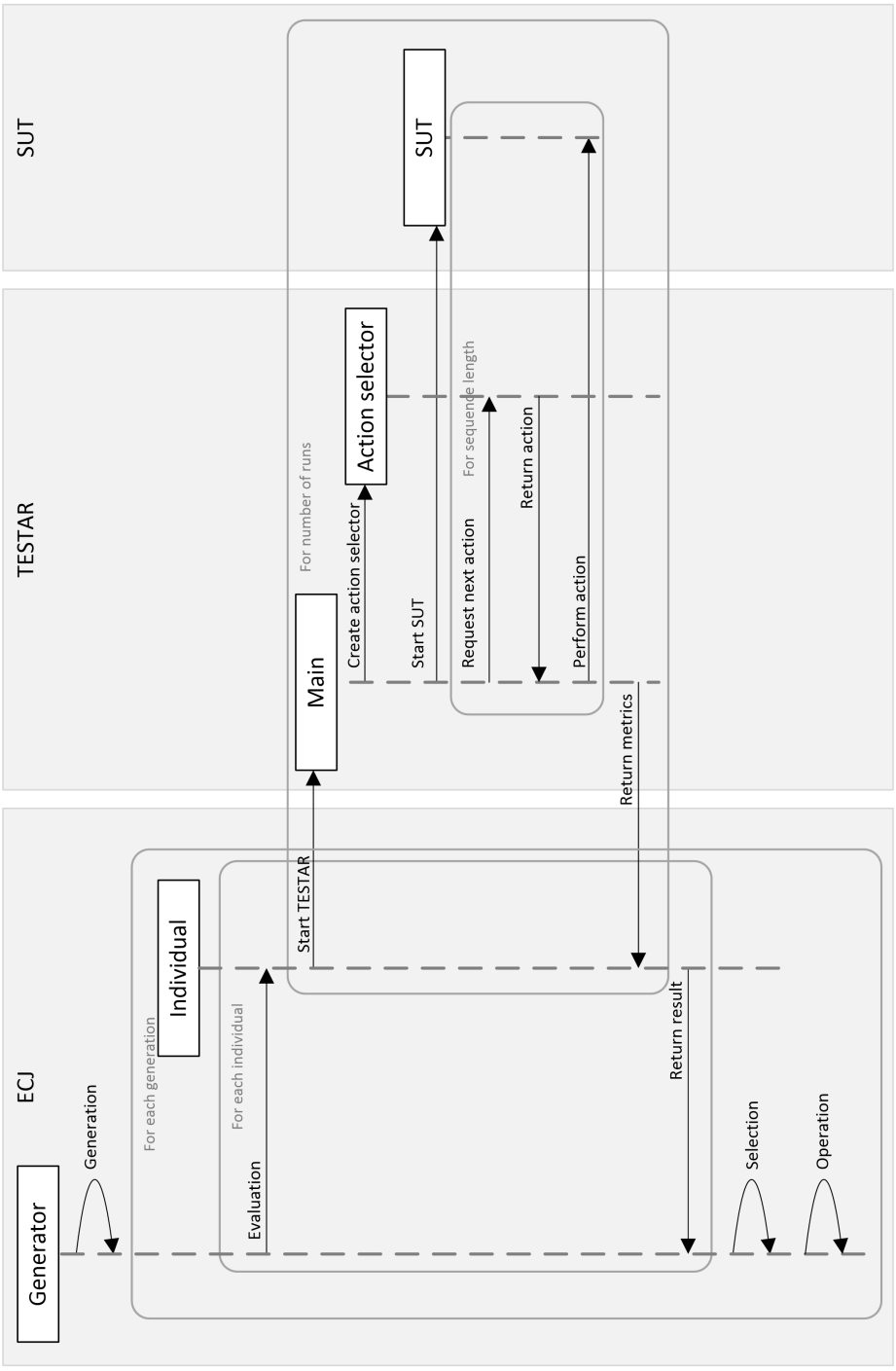

Figure 5.2: This sequence diagram shows how TESTAR is integrated in the evolution of ECJ.

reached for the next generation. In the random creation of individuals for the first generation, chances are that duplicates occur. ECJ will try a configurable number of times to create a unique individual. This parameter, *duplicate retries*, is set to 100 by default. If no unique individual has been found after that number of attempts the duplicate individual is accepted. The generated trees could in theory be of infinite size. A *maximum tree depth* can be configured to limit the size.

5.2.3 Evaluation

In the evaluation phase TESTAR is run. TESTAR performs a configured number of actions on the SUT in each run, known as the *sequence length*. Furthermore, for each evaluation a *number of runs* with TESTAR can configured. Since many strategies will have a large random component, strategies can be lucky in a run or an action selection. Short runs or small numbers of runs are therefore not representative for the expected performance of that strategy in long or frequent runs. The fitness should therefore be based on the metrics of multiple runs. The more often TESTAR is run for each strategy and the longer the sequences, the longer the evaluation obviously takes. An optimum needs to be found for the sequence length and the number of runs.

Each run results in *metrics*, which can be used as input for the *fitness function*. This fitness function needs to be configured, determining which metrics are used as input and how these are used to calculate a fitness value. Better individuals should have a fitness closer to zero, while worse individuals should have a higher fitness value, as the Koza-style evolution dictates.

5.2.4 Selection

As in biological evolution, the chance of survival is a combination of luck and fitness. The standard *selection mechanism* of Koza is tournament selection. A random set of individuals is picked, and the fittest individual of this set is then selected. This is repeated a number of times until the configured population size for the new generation is reached. The *tournament size*, the number of individuals selected per tournament, can be configured. The ECJ default is two, while the Koza default is seven. A larger tournament size increases the chance of the fittest individual of the population to be selected.

Another parameter that can be considered a selection parameter is *elitism*. The *elite*, a configurable number of the top fittest individuals, survive to the next generation without tournaments or operations. In ECJ the elite individuals are not re-evaluated. They maintain their fitness value.

5.2.5 Operation

The selected individuals are run through an *operation pipeline* to create new individuals for the next generation. ECJ provides several methods for *operations* that are explained below and visualized in Figures 5.3 - 5.7.

Reproduction recreates the same individuals

Mutation takes one node including its subtree from an individual, and replaces it with a randomly generated tree of the same return type.

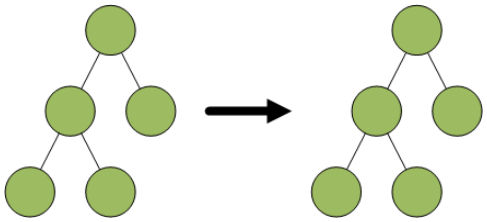

Figure 5.3: Reproduction: recreates the same individuals

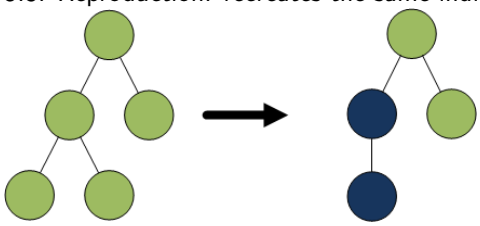

Figure 5.4: Mutation: takes one node including its subtree from an individual, and replaces it with a randomly generated tree of the same return type.

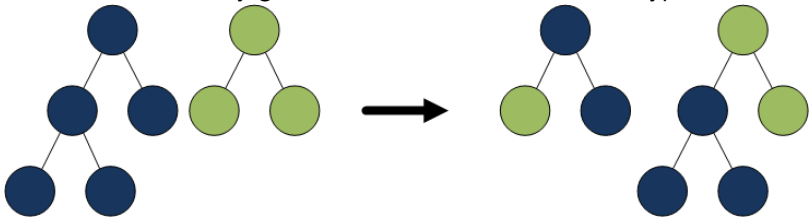

Figure 5.5: Cross-over: exchanges nodes of the same return type, including their subtree, between selected individuals. Both individuals are returned.

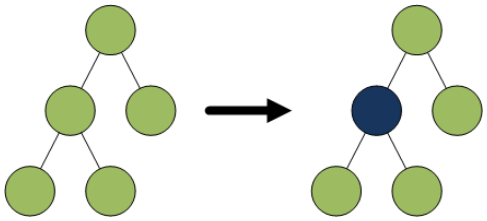

Figure 5.6: Mutate-one: takes one node from an individual, and replaces it with a random node with the same arity and type constraints.

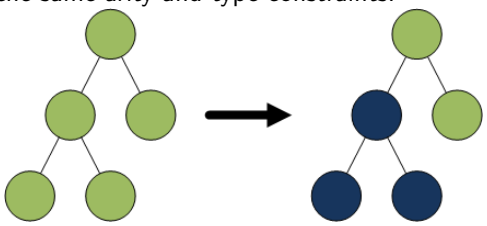

Figure 5.7: Mutate-all: takes one node from an individual, and replaces all nodes in its subtree with random nodes with the same arity and type constraints.

Cross-over exchanges nodes of the same return type, including their subtree, between selected individuals. Both individuals are returned.

Mutate-one takes one node from an individual, and replaces it with a random node with the same arity and type constraints. The child nodes of the mutated node are maintained.

Mutate-all takes one node from an individual, and replaces all nodes in its subtree with random nodes with the same arity and type constraints. The structure of the tree is maintained.

The default operations for Koza are *reproduction* and *cross-over*. It is important to note that both reproduction and cross-over only use the components that are present in the selected individuals. Therefore with default Koza settings the number of possible variations is reduced with every generation. Also note that some nodes are unique in their arity and type constraints, for example the if-statement, so they cannot be replaced in the mutate-one or mutate-all pipeline.

5.3 Choosing the right settings

This section describes the values and settings chosen for the previously described parameters and gives a motivation. The two aspects that occur regularly in the motivation are *time* and *reliability*. The aspect *time* refers to the amount of time needed for the entire evolution. If the evolution takes ages due to its settings, fewer experiments can be run within the time of the research. The aspect *reliability* is important because in the described setup there are many factors of randomness. The individuals are randomly created, the strategy selects random actions and the selection phase selects individuals at random for the tournament. For the purpose of time, short runs and small populations are practical, while many long runs and a large population increase the reliability of the results. Choosing the right parameter settings requires balancing between these two aspects.

5.3.1 General parameters

As mentioned in the previous section, the generational model of evolution is used, with the Koza tree-based setup. The number of *generations* has been set to ten. The later generations of those ten already show stabilization of the resulting strategy set in trial runs. In most runs the best individual of a generation was the same for each generation after the fifth. Therefore more generations are deemed not to contribute much to the result.

Three systems under test (SUTs) are selected for the experiments: Windows 7 Calculator, VLC Media Player [41] and Testona [40]. Windows 7 Calculator is chosen because it is readily available and easy to use. Testona is also used in EARV [17]. VLC Media Player is easy to set up with TESTAR and is used in a parallel research. The three systems are also different from each other in prevalence of controls and action types. For reliability of the results, four evolution runs are done per SUT.

5.3.2 Creation parameters

The *population size* in this research has been set to 100. This is a pragmatic choice. The number is large enough for diversity (contributing to reliability) but

small enough to be able to evaluate all individuals in a reasonable amount of time. More on the duration of the evaluation is explained in the following subsections.

The number of *duplicate retries*, retrying to generate a unique individual if a duplicate was created, is by default set to 100. This value has not been changed, as generating individuals costs very little time while having a generation of only distinct individuals increases the reliability of the results. The *maximum tree depth* has been set to the default value of 17. This already allows for an immense range of strategies, while deeper trees are not expected to yield better or worse results. The tree depth does not noticeably influence evolution time.

5.3.3 Evaluation parameters

In the evaluation phase, TESTAR will be run with the strategy of the individual to determine its fitness. The parameters that need to be configured for this phase are the *sequence length* of each run, the *number of runs*, the *metrics* of the runs used to calculate the fitness, and the way the fitness is calculated from the selected metrics in the *fitness function*.

Sequence length and number of runs

Because of the large random factor in the test runs, results will vary greatly between lucky and unlucky runs. To eliminate the *luck factor* it would be desirable to execute as many runs as possible, with as many actions as possible. This would take a very long time. To analyze the impact of the luck factor, trial runs with TESTAR are done with a random list of strategies. One of the available metrics, the number of states visited, is compared across runs of the same strategy.

Figure 5.8 shows the distribution of the number of states visited per run in these trial runs on the Calculator. Each box is a strategy, which is run 20 times with sequence length 100. The strategies are sorted by average number of states visited, from low to high, where high values are considered better. It is clear that the first three strategies score significantly worse than the others, but the spread of results of all following strategies is very large. In fact, the best run of the fourth strategy is better than the average of the right-most, best strategy.

Figure 5.9 shows how much the average number of states after x runs (on the x-axis) deviates from the average number of states of the same strategy over 20 runs. This graph is based on the same data as Figure 5.8. After the first run, the result is on average almost 90% higher or lower than the average result after 20 runs. Both these graphs suggest a strong luck factor, as expected.

Can the luck factor be decreased with longer runs? Figures 5.10 and 5.11 show the same distribution graph with runs of sequence lengths 200 and 20. This shows that the issue of a large distribution is not eliminated with longer runs. Instead, longer runs allow a greater variety in the number of states visited, as the number of states is maximized by the sequence length. This is because the application can only transition to another state by an action, while not all actions cause the application to go to another state. A broader range allows a larger deviation.

A possibility is to run a strategy a small number of times per evaluation, run it again when it reoccurs and calculate a fitness based on the metrics of both the previous and the new runs. This would mean that a strategy that was lucky in a single run has an advantage over a strategy that was run many times with only a few lucky runs. Therefore the choice is made to run each strategy the same number

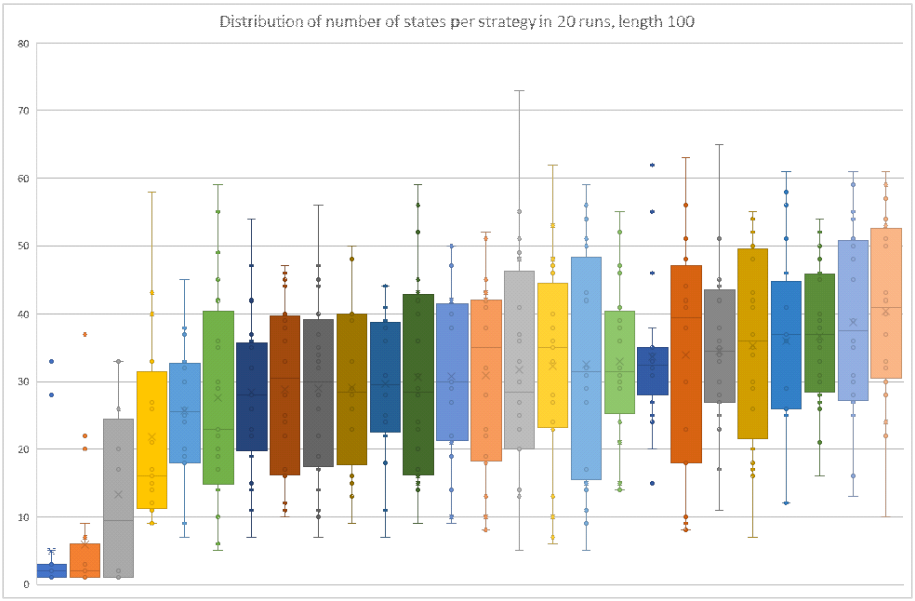

Figure 5.8: The distribution of the number of states visited per run with the same strategy. Each box is a strategy. Each strategy is run 20 times, with 100 actions.

of times during each evaluation and not run it again when it reoccurs. To balance between reliability and time, each strategy is executed 20 times with 100 actions.

Metrics and fitness function
The metrics that TESTAR provides as introduced in Chapter 2 are listed below. Some are based on the state transition model that TESTAR generates, others are otherwise observed by TESTAR.

Graph states the number of unique states in the state transition model

Graph actions the number of unique actions that have been executed, shown as transitions in the model

Test actions the total number of non-unique actions that have been executed

Abstract states the number of unique abstract states visited, in which an abstract state is a group of states that are the same when a certain set of properties is disregarded (for example the color of controls)

Longest path the length of the longest path in the state transition model

State coverage the minimum and maximum state coverage, defined as the lowest and highest rate of executed actions over the total number of available actions of a state

Memory usage the peak amount of memory used by the SUT

CPU usage the peak amount of CPU used by the SUT

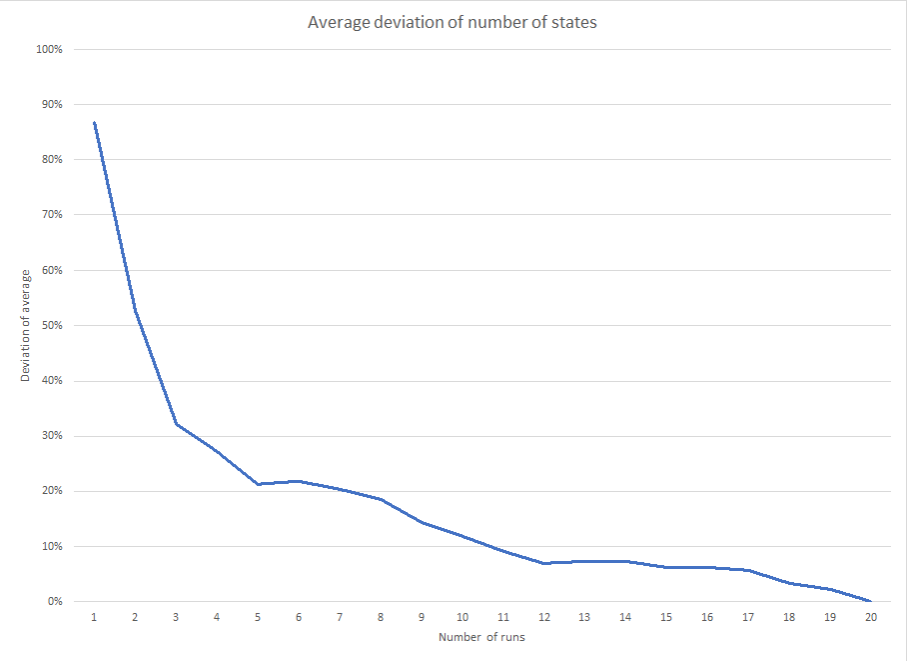

Figure 5.9: The deviation from the average after each run, compared to the average after 20 runs.

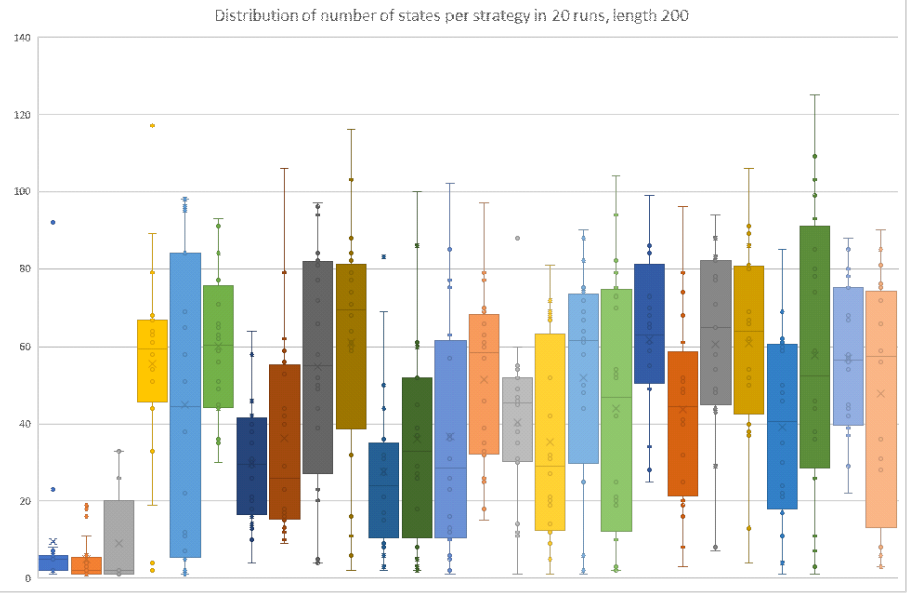

Figure 5.10: The distribution of the number of states visited per run with the same strategy, with 200 actions per run.

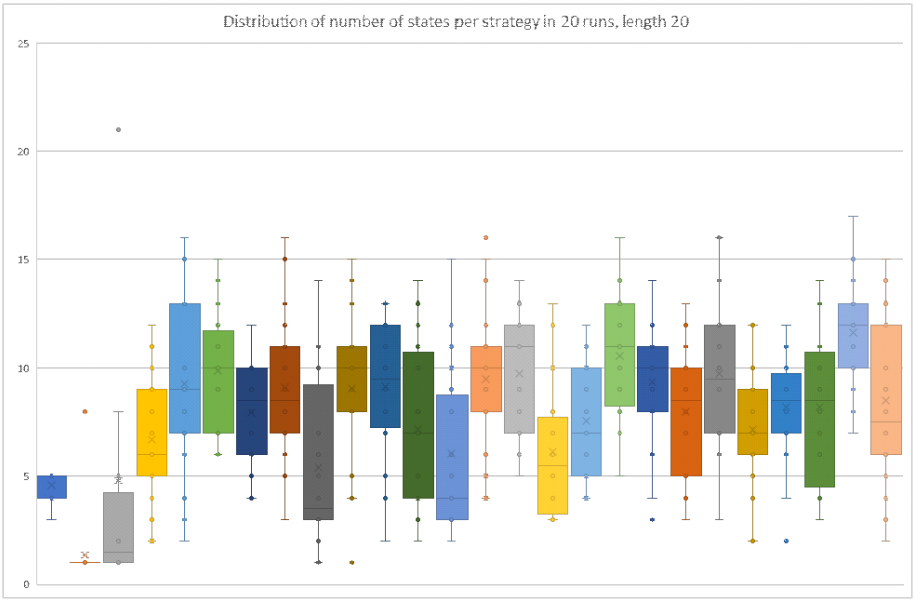

Figure 5.11: The distribution of the number of states visited per run with the same strategy, with 20 actions per run.

Faults the number of faults detected by TESTAR, according to the configured oracles

Some of these metrics are not considered suitable as an input for fitness. An example is the number of abstract states, which is used in EARV. Two visited states can be considered the same abstract state, for example if it is the same screen with the same set of controls, but with different text in a text field. The text property can be disregarded in comparing states, leading to the conclusion that they are the same abstract states. Which properties of the controls TESTAR should disregard in determining abstract states requires more research, so this metric is not used. Recommendations for further research regarding this metric are included in Chapter 8.

Other metrics, for example the number of detected faults, are not reliable due to their low occurrence. If there are only a few bugs in the application, the chance of finding them in the limited sequence length of an evolutionary run is too small to be able to judge whether it was found because of the strategy or because of sheer luck.

Then there are possible metrics that are currently not measured by TESTAR, like the code coverage. In another project [29] the code coverage of TESTAR's tests is measured with the code coverage tool JaCoCo [23]. Implementing this tool in the evolution flow would cost extra time, which does not fit in this research. Furthermore it is expected that the percentage of code covered during a relatively short sequence is too low to be a significant metric. Therefore it is not implemented in the current project. It is included in the recommendations for further research.

Metrics of the strategy tree itself could also be used as input, for example size. Should a shorter strategy be preferred over a longer one? A shorter strategy may look more sensible, but in the end it is about how well it tests. A strategy with

a useless looking subtree like 'number of actions equals number of actions' may not seem sensible, but in a next evolution round one of the nodes may be replaced with another node, resulting in a strategy that does make sense. Furthermore, the useless subtree does not mean that the strategy is useless. Therefore the length of the strategy or the presence of useless subtrees should not influence the fitness value.

A pragmatic choice has been made for the metrics used in the fitness function: metrics that are available and reliable, and that are likely to predict how well a SUT is tested. The first used metric is the number of graph states, as this metric clearly measures the exploration of the SUT. The second metric is the length of the longest path, as this metric measures how deep a SUT is explored.

The optimal fitness according to Koza is zero, so a calculation is done with the two values to get a fitness value closer to zero when these numbers are higher. Simply dividing one by the sum of the two values would result in the fitness values quickly decreasing between low values, while being very close to each other between high values. To spread out the results more evenly and on a scale from zero to ten, the square root of the sum is taken, and the total is multiplied by ten. This results in the following fitness function.

$$fitness = \frac{10}{\sqrt{graphstates + longestpath}}$$

Because multiple runs are done, the medians of the number of graph states and the length of the longest path of all runs are used to calculate the fitness value of the strategy. If the average would have been used instead of the median, outliers would have had a too large effect on the fitness.

5.3.4 Selection parameters

As described in the previous section, selection of individuals is based on two components: *fitness* and *luck*. The *tournament selection* method is used, which is the default for Koza. The default tournament size is seven. This means that seven individuals are selected at random from the entire population (*luck*), and then the *fittest* individual of these seven is selected and processed in the operation pipeline. The tournament size is increased to 15 in this research, as fitness is deemed relatively important in the context of this research, and the population size is much larger than in EARV (100 vs 20), where the tournament size was 5. No elaborate analysis has been done to compare different settings for this parameter.

To make sure that the best individuals make it to the next generation, without needing to be lucky or being mutated, an elite is used. An elite size of 10 is chosen, meaning that the best 10 individuals are taken to the next generation without changes. The following subsection elaborates more on the effect of using an elite.

5.3.5 Operation parameters

The different operation pipelines as introduced before (crossover, reproduction, mutation, mutate-one and mutate-all) and whether or not elitism is used, yield different results in evolution. Some pipelines result in individuals that are largely the same as the ones in the previous generations, other pipelines create completely different individuals. The number of distinct individuals in the first generation is the configured population size, unless duplicates occurred after the set number of duplicate

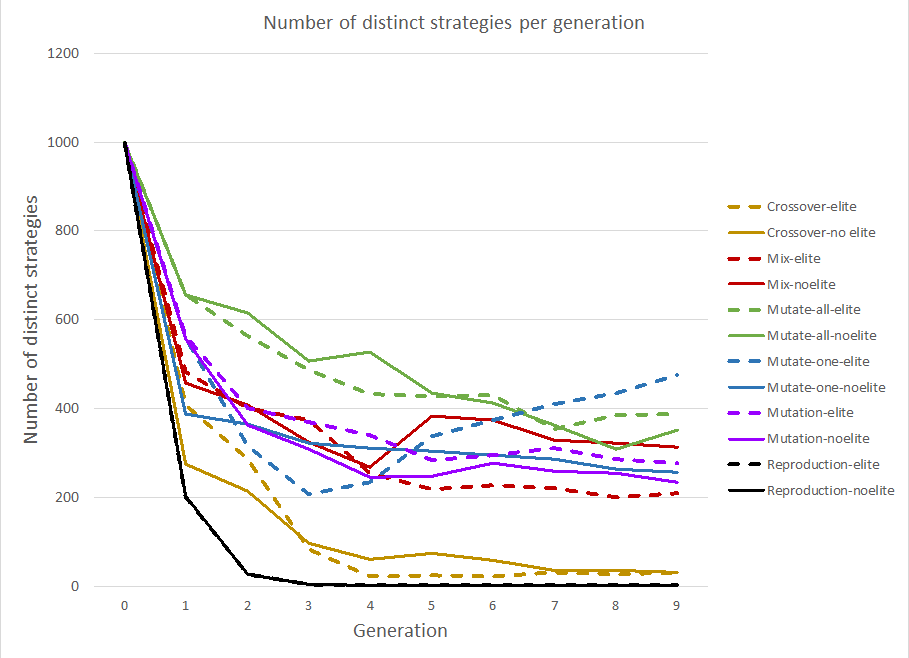

Figure 5.12: The number of distinct individuals in each generation, with different pipelines and an elite of 10 or 0.

retries. The number of distinct individuals decreases with each generation, as the individuals depend on the surviving individuals of the previous generation, which is a subset of all individuals in that generation. With some pipelines the decrease happens quicker than with other pipelines.

These factors are important in the attempt to find *the* fittest strategy, because when the same individuals are used in each generation, the final fittest individual greatly depends on the individuals randomly generated in the initial generation. When new individuals occur in every generation, more strategies are tried and the chance increases that the fittest possible strategies are among them. Therefore the pipelines are analyzed more closely and compared on the diversity of strategies, so that an optimal configuration can be found.

To compare the pipelines trial evolutions are run with each of the pipelines separately. The generation size to analyze the effect should be large, to have more reliable data. This would take weeks if it were done using TESTAR and an actual SUT. Since the primary interest is in the number of distinct strategies, a random fitness value is assigned in the evaluation phase instead of running TESTAR. Several evolution runs with large generations can then be run in a few minutes. The results are explained below.

First, the number of distinct strategies in each generation is analyzed. This means that the duplicates within a generation are not counted. The results are shown in Figure 5.12. There clearly is a difference between pipelines. The dashed lines are with an elite of 10, the solid lines are without elite. The crossover and reproduction pipelines clearly yield the smallest amount of distinct strategies in each generation, while the mutation pipeline yields the largest amount.

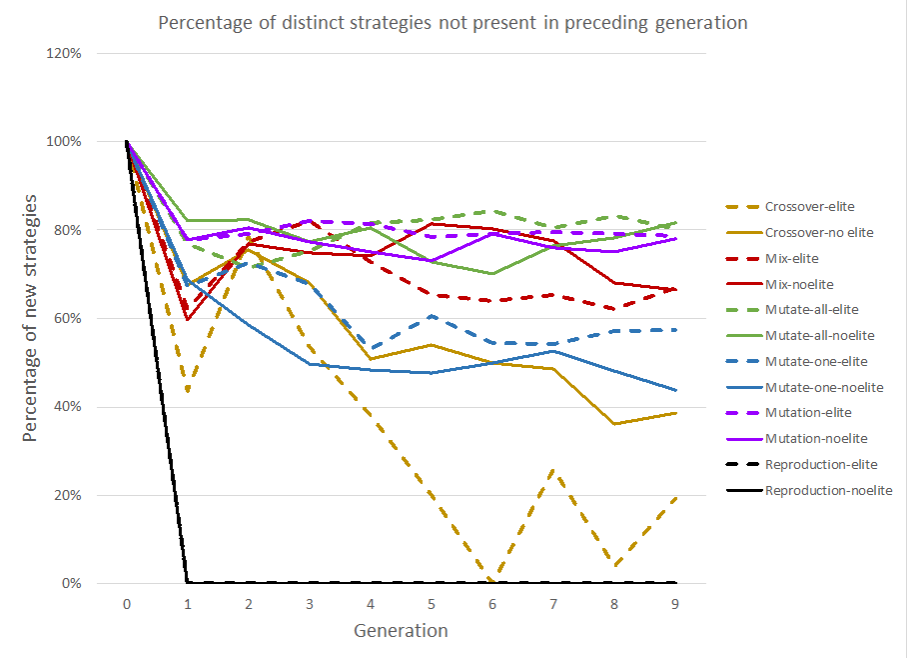

Figure 5.13: The percentage of distinct individuals in each generation that did not occur in the previous generation, with different pipelines and an elite of 10 or 0.

Another interesting aspect is how many of these distinct individuals did not already occur in the previous generation. Figure 5.13 shows the percentage of the distinct individuals in each generation that are new compared to the immediately preceding generation. Again the crossover and reproduction pipelines score lowest, indicating that many of its individuals already occurred in the preceding generation.

To have a balance between new and existing individuals, and to benefit from the different ways individuals are mutated with different pipelines, a mix of all pipelines is chosen. As can be seen in the two previous graphs, the mix scores average to high on the number of distinct strategies. In the mixed pipeline, each pipeline has a probability of being used. The probability for each pipeline is set to the same value of 20%, so that the distribution over the pipelines is equal. Furthermore, an elite of 10 is used to make sure that the best strategies of every generation survive to the next.

5.3.6 Optimizing evolution runs

A single TESTAR run of sequence length 100 can take approximately two minutes, depending on the SUT. Evaluating a single individual 20 times will then take 40 minutes. A generation of 100 individuals would then take 67 hours, almost three full days. It is clearly worth looking for measures to save time without losing results or reliability. Two time-saving measures are used: reusing results of previous evolutions and reusing results for logically equivalent strategies.

The metrics of each run of a strategy are stored in ECJ during the evolution runs. When the strategy reoccurs in another individual, it is simply given a fitness

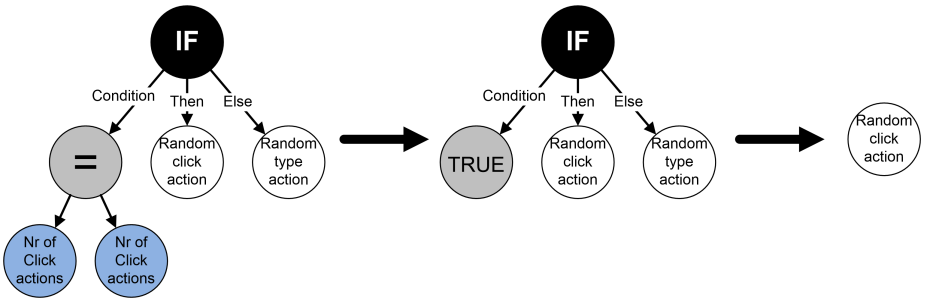

Figure 5.14: The simplification process.

calculated from the previous results, without running it again. This not only saves time, it also ensures that the strategies are evenly compared. To save more time when running multiple evolutions with the same SUT and sequence length, the metrics of all runs of all previous evolutions are reused. These metrics are first loaded into ECJ before a new evolution starts. When a strategy is encountered that was already run in a previous evolution, the metrics of the previous runs are used and TESTAR does not need to be started. This is also helpful when evolutions are ended unexpectedly, for example when the monkey test deletes files that are needed to run the software.

Some strategies are not the same, but are logically equivalent. For example if the boolean of an if-statement is always true, the if-true action is always selected. This makes the strategy equivalent to a strategy that consists only of the tree in the if-true subtree. Therefore, to save time, strategies are simplified to a logically equivalent form as far as possible, as shown in Figure 5.14. Both sides of the 'equals' tree are the same, so the statement will always evaluate to true. The strategy can therefore be simplified to the action of the true-branch of the if-statement.

The results of a strategy are stored under the name of the simplified version of that strategy. That way logically equivalent strategies are treated as the same *canonical strategy* when evaluating them. This saves time, as TESTAR is only run the maximum number of times in total for a canonical strategy.

Canonical strategies are only equivalent for TESTAR, not for ECJ. A different tree can result in non-equivalent strategies after mutations. Therefore the tree that ECJ uses for the evolution is not changed during simplification.

5.4 Summary

With the parameter settings summarized in Table 5.1 evolutions are run using the three different SUTs. Four evolutions are run on each SUT, with 10 generations, 100 individuals per generation, running each canonical strategy 20 times with a sequence length of 100. An archive is created of all previous runs, that is loaded into ECJ before the next evolution starts.

Table 5.1: Summary of the parameters and the chosen settings, compared to those of EARV [17] if applicable

Parameter	Setting	EARV Setting
	General	
Evolutionary model	Generational	Steady state
Method	Koza (tree-based)	Koza (tree-based)
Generations	10	*N/A*
SUTs for evolution	Calculator, VLC, Testona	PowerPoint
SUTs for comparison	Calculator, VLC, Testona	PowerPoint, Testona, Odoo
Evolution runs	3×4	1
	Creation	
Population size	100	20
Duplicate retries	100	*N/A*
Max tree size	Depth: 17	Node count: 20
	Evaluation	
Sequence length	100	100
Number of runs	20	1
Metrics	Graph states, longest path	Abstract states
Fitness function	$\frac{10}{\sqrt{graphstates + longestpath}}$	$abstractstates$
	Selection	
Selection method	Tournament selection	Tournament selection
Tournament size	15	5
Elite	10	*N/A*
	Operation	
Operation pipelines	Mutation, Mutate-one, Mutate-all, Crossover, Reproduction	Mutation and Crossover

Chapter 6

Architecture

This chapter describes the architecture of the Java classes that were added to both ECJ and TESTAR for this research. Only the added or changed classes are described and the class diagrams are slightly simplified, omitting irrelevant methods and parameters. The ECJ classes needed for the evolution are described in Section 6.1. The TESTAR classes needed for the interpretation and use of the strategies are described in Section 6.2.

6.1 ECJ classes

ECJ needs to be extended in order to generate the desired trees of strategies and to evaluate them by running TESTAR. Figure 6.1 shows the part of the architecture related to the strategy trees. ECJ has an Individual class that contains a tree. This tree refers to the top node of the tree, and each node can contain other nodes. The node types are created by creating classes extending the existing abstract GPNode class. The return type and the child types of each node are not defined in these classes, but in a parameter file of ECJ.

For this research, the original GPNode class is extended with an abstract StrategyNode class, as shown in Figure 6.1. StrategyNode contains methods that are needed specifically for the strategies. The method *toFullString* creates a string that can be sent to TESTAR, and the methods *getSimplifiedTree* and *getSimplifiedNode* are used to simplify the strategy to get the canonical strategy. For each action selection node type as listed in Chapter 4 a class is created that extends StrategyNode. These classes only contain their specific name, short name (for quick reference) and required number of children. Some classes contain a specific method for the simplification process. The return types and the child types of each node class are defined in the ECJ parameter file in accordance with the lists in Chapter 4. That is all the information ECJ needs to generate individuals with trees of nodes, which represent strategies.

When evaluating an individual, ECJ calls an object of class GPProblem. This class has a setup method and an evaluate method. The evaluate method has an Individual object as input and sets the fitness value of that individual after evaluating it.

Evaluation is done by running TESTAR. A TestarProblem class is created that extends GPProblem, as shown in Figure 6.2. The setup method reads parameters

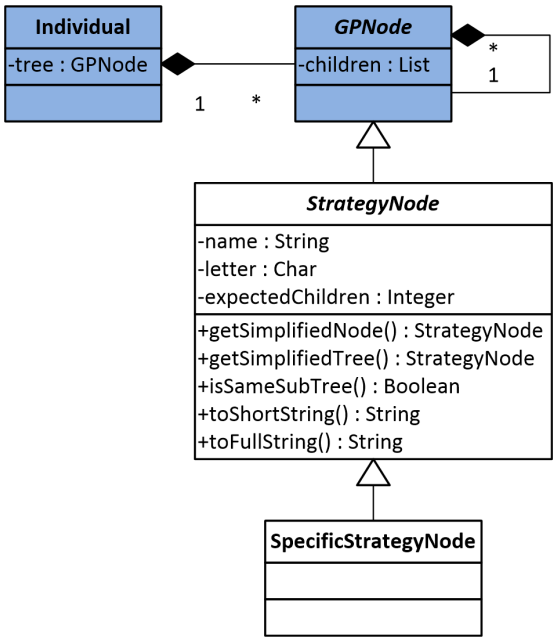

Figure 6.1: The architecture of ECJ: nodes. The original ECJ classes are blue, the other classes are created for this research.

from a file, for example the sequence length and the desired number of runs, so that these can be configured easily. The evaluate method distills the strategy from the tree of the individual and asks an Evaluator object to evaluate it. The Evaluator uses a TestarRunner object to run TESTAR. TestarRunner gets the strategy as a string, writes it into a file that TESTAR uses as input using the StrategyWriter, runs TESTAR and waits for it to finish.

When TESTAR is finished, the results are read from a file that TESTAR outputs, using the ResultsReader. The Evaluator asks for these results, stores them and writes them into a file using the ResultsWriter for later analysis. It then asks the TestarRunner to run TESTAR again until the desired number of runs is reached. The Evaluator then returns the fitness calculated over the results of all runs to the TestarProblem, which then sets the fitness value of the individual. The progress is monitored using a ControlWindow, shown in Figure 6.3. This is a simple GUI showing information on the current run and with buttons to pause or stop the process.

The classes Strategy and Result (see Figure 6.4) are used in the communication between other classes. The difference between an Individual and a Strategy is that an Individual is an entity as used by ECJ for the evolution, while a Strategy is a distinct action selection strategy. An Individual contains a tree that represents a strategy, and different individuals can contain a tree representing the same strategy. A strategy can have several representations, including a textual form (used for legibility), a short form (used for reference in later analyses), a canonical form in which the simplification steps are applied as described in Subsection 5.3.6 and a short canonical form.

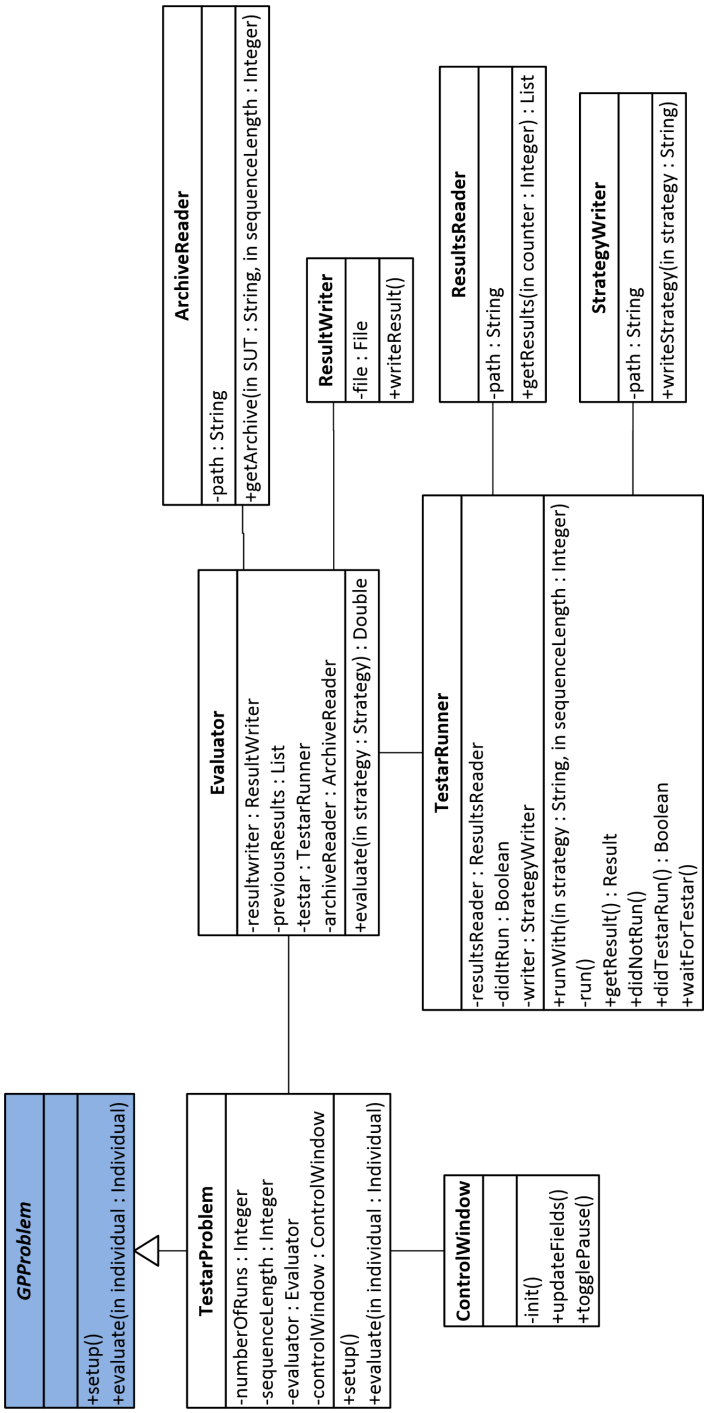

Figure 6.2: The architecture of ECJ: evaluation

Figure 6.3: The ControlWindow with which the progress can be monitored

The constructor of the Strategy class takes the StrategyNode tree of the individual as input. It then distills the textual and short form from it, simplifies the tree and distills the textual and short form of the simplified tree. After this, the Strategy object contains the four strings. The tree is not stored in the object.

The run results are saved and managed in a Result object. The Evaluator object contains a previousResults object of type TreeMap. This is a list of key-value pairs, with the canonical strategy as the key and a Result object as the value. All result values of all the runs of a canonical strategy are stored by the Result object. When the Result is asked for the fitness, the fitness is calculated using the fitness function and returned.

One of the time saving measures as described in Section 5.3.6 is to evaluate the canonical strategy only once. When the Evaluator is asked by TestarProblem to evaluate an individual, it first checks whether the canonical strategy is already in the previousResults, and if so, whether the maximum number of runs has been reached. If it is, the fitness based on these runs is directly returned without evaluating the individual. If not, the individual is evaluated and the results of the runs are added to the list.

The other time saving measure is to reuse results of previous evolutions. After an evolution run is finished, the results are copied to an archive file for a specific SUT and sequence length. When ECJ is started again and the Evaluator is initialized, the ArchiveReader object reads the previous results from the appropriate archive file. These results are then stored in the aforementioned previousResults list, the same as where the result of the current evolution run will be stored. A strategy that is the same as or equivalent to a strategy in the previousResults is not reevaluated.

Besides running evolutions with ECJ, specific lists of strategies need to be run for further analysis in the second part of the experiments. Many of the same components are needed as for the evolutions. Therefore a class is added, the StrategySetRunner, shown in Figure 6.5, with its own main method. The StrategySetRunner reads a list of strategy strings from a file using the StrategyReader. The strings are

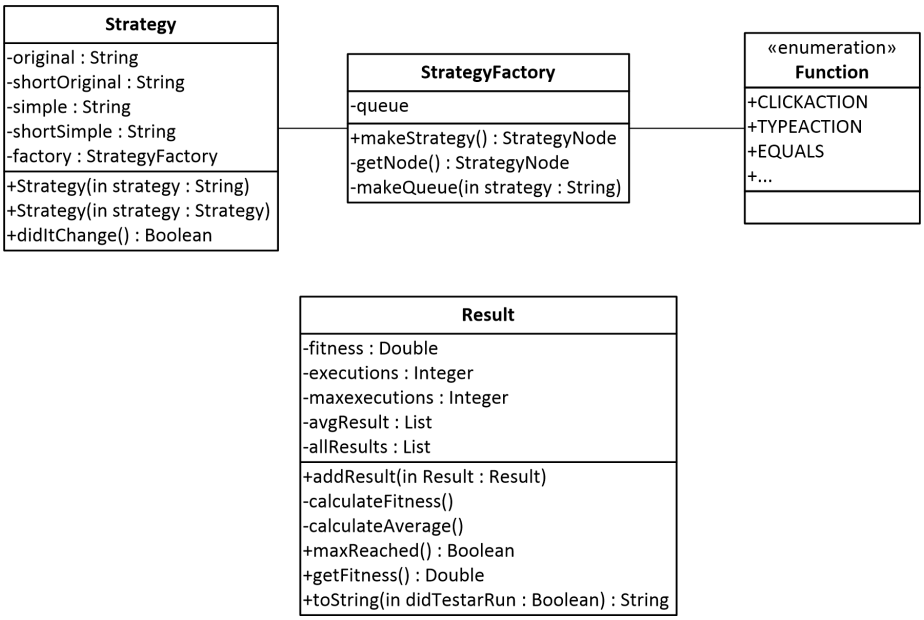

Figure 6.4: The Strategy and Result classes

converted back into trees to be able to distill the different forms that the Strategy object needs. For this the Strategy object has a second constructor that takes a string as an input. From this string a tree is built using the static StrategyFactory in the same way as is done in TESTAR, which is explained more elaborately in the next section.

6.2 TESTAR classes

In the existing architecture TESTAR creates a UIWalker object when it starts, which is used to select the next action. When the next action needs to be selected, the UIWalker gets all information on the current state and returns an action that TESTAR then performs on the SUT. TESTAR already contains an AbstractWalker class and several Walker classes extending this abstract class, that have been used in previous experiments.

For this research a StrategyWalker class extending the AbstractWalker class is created, as shown in Figure 6.6. When ECJ starts TESTAR, a commandline parameter is passed telling TESTAR to run with a tree-based strategy, so that TESTAR uses the StrategyWalker. During the setup, a SelectorFactory object is created, as shown in Figure 6.7. The SelectorFactory reads the strategy from the file in which it is written by ECJ. As this is a string that represents a tree, the SelectorFactory uses a recursive method that transforms the string element by element back into a tree.

The SelectorFactory first uses the makeQueue method that cuts up the strategy string into its node elements, replaces the elements by their equivalent from the Function enumerator and puts them in a queue. Then the getNode method is called, which takes and removes the first element of the queue and returns a node

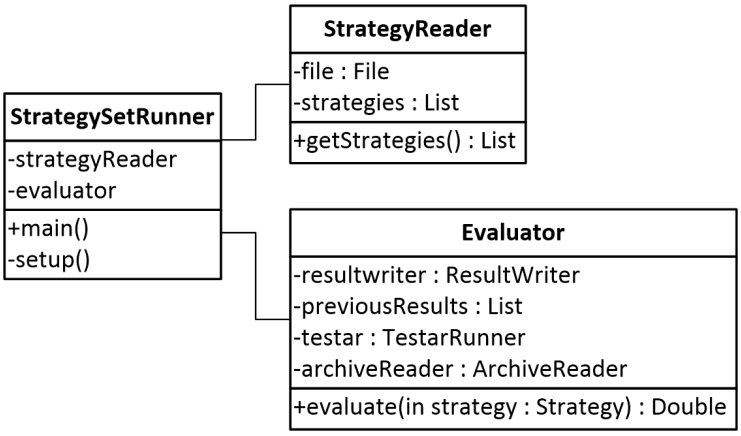

Figure 6.5: The StrategySetRunner class

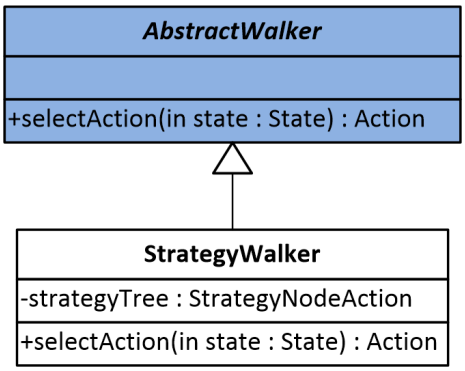

Figure 6.6: The StrategyWalker class extending the AbstractWalker class

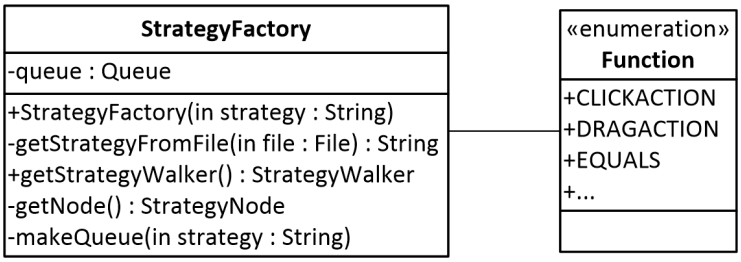

Figure 6.7: The StrategyFactory class with the Function enumerator

object of the right class, using a switch case statement. If the node type requires children, the getNode method first calls itself for each of the expected children, and the retrieved child nodes are added to the list of children of the current node before it is returned. This results in the same tree structure as ECJ generated.

Figure 6.8 shows the class hierarchy for the StrategyNode classes. TESTAR has to make a distinction between nodes based on their return type, as each type has its specific method. As listed in Chapter 4 the possible return types are Action, Actiontype, Boolean and Number, so for each of these an abstract class is created that extends the abstract StrategyNode class. Node classes for each of the node types listed in Chapter 4 are created, extending the StrategyNode class of the right type. Each of the node classes implements a method to get the result of the node function. For example, the RandomAction class returns a random action based on the current state, while the Equals class compares the return values of its two children and returns true or false. The top node of the tree eventually returns an action. In case the strategy results in no selected action, for example when it tries to select a type action while there are no type actions, it returns null. The StrategyWalker then selects a random action, so that TESTAR can continue, and it reports the number of times this happens for later analysis.

In order to minimize the changes to the existing classes of TESTAR, a StateManager class is created to implement all methods needed for selecting actions and getting other properties of the current state. As shown in Figure 6.9 the class implements a long list of methods, many of which are overloaded, using different parameters. The StrategyNodes use these methods to get their return value.

With the described new classes in both ECJ and TESTAR, the software is ready to run evolutions, test with the new strategies and evaluate their performance.

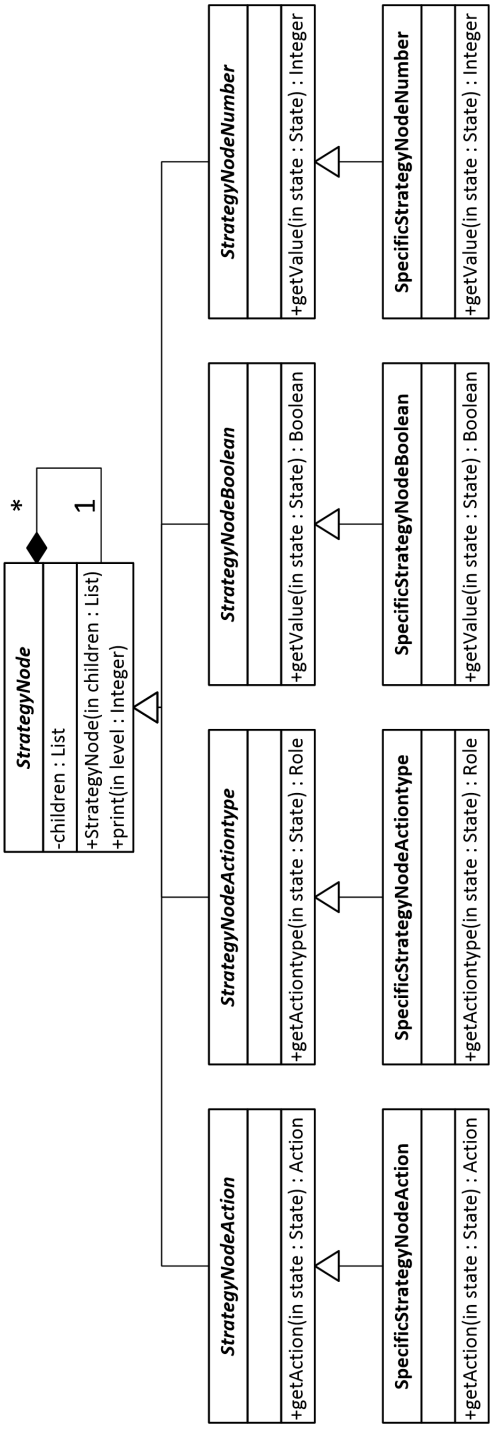

Figure 6.8: The StrategyNode hierarchy

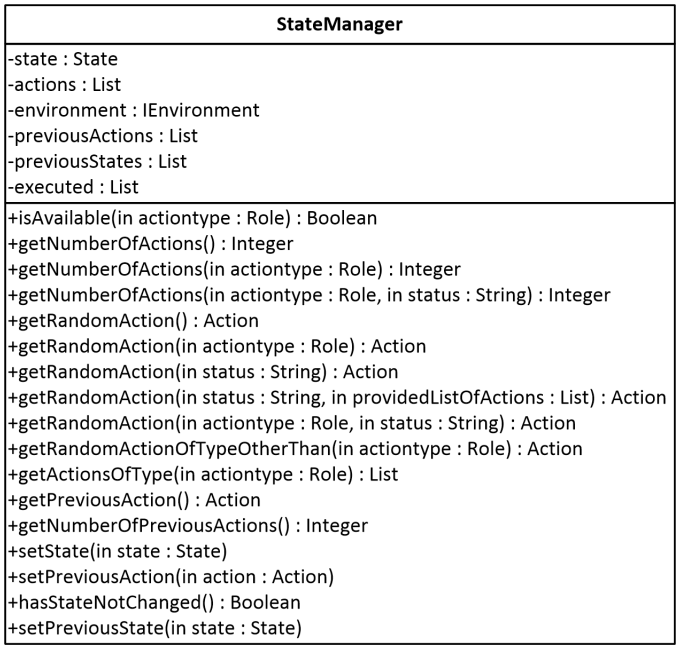

Figure 6.9: The StateManager class

Part III

Experiments and conclusions

Chapter 7

Results

As explained in Chapter 3 and shown in Figure 7.1, the experiments are executed in two phases: Phase A comprises of evolution runs and phase B comprises of tests with the best strategies of the evolutions. In phase A evolutions are run using three different systems under test (SUT): Windows 7 calculator, VLC Media Player [41] and Testona [40]. Four evolutions are done on each SUT, with 10 generations, 100 individuals per generation, running each canonical strategy 20 times with a sequence length of 100 actions. Section 7.1 lists the fittest strategies of each of these evolution runs.

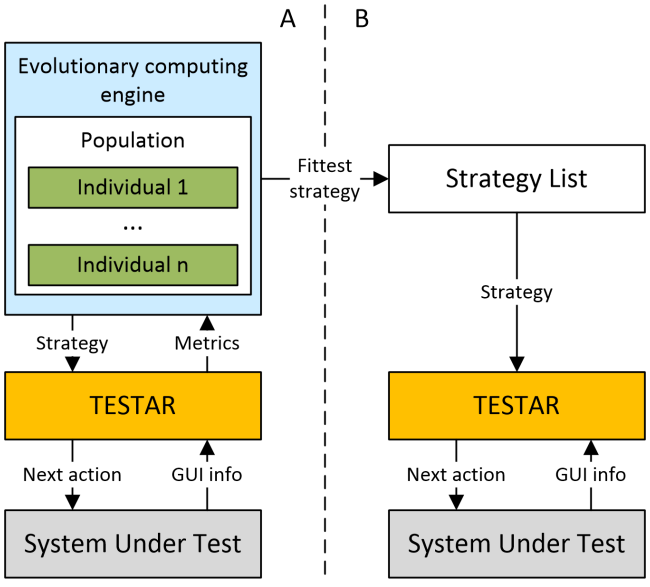

Figure 7.1: The setup of the research, with the evolution on the left and the tests for further analysis on the right.

To be able to answer the research questions stated in Chapter 3 the performance of the found fittest strategies needs to be compared with the performance of the random strategy and the EARV strategy. Furthermore all these strategies need to be run on all three SUTs, in order to make comparisons across SUTs. Therefore

in phase B, tests are run using TESTAR with each of the found fittest strategies. The test runs are done with sequence length 1000 and 100, 20 times each, for each strategy on each of the SUTs. Section 7.2 analyzes the results of these experiments, to accept or reject the hypotheses stated in Chapter 3.

7.1 Phase A: Evolution

Each of the evolutions yields one fittest strategy, defined as the canonical strategy with the best fitness value in the last generation. Below the fittest strategies of the evolution runs are listed per SUT.

Calculator
The four evolution runs done on the Calculator resulted in only two unique fittest strategies due to duplicates. The strategies are shown in Algorithms 7.1 and 7.2.

Algorithm 7.1 Fittest strategy of Calculator 1

```
random-action-of-type-other-than
    type-of-action-of
        If type-actions-available Then
            random-most-executed-action
        Else
            random-action
        EndIf
```

Algorithm 7.2 Fittest strategy of Calculator 2

```
random-unexecuted-action-of-type
    type-of-action-of
        previous-action
```

VLC Media Player
The same as for the Calculator, due to duplicates only two unique fittest strategies resulted from the evolutions on VLC Media Player. These are shown in Algorithm 7.3 and 7.4.

Algorithm 7.3 Fittest strategy of VLC 1

```
random-unexecuted-action-of-type
    type-of-action-of
        random-least-executed-action
```

Algorithm 7.4 Fittest strategy of VLC 2

```
random-unexecuted-action-of-type
    type-of-action-of
        random-action-of-type-other-than
            type-action
```

Testona

The four evolution runs on Testona resulted in four fittest strategies, shown in Algorithm 7.5 to 7.8.

Algorithm 7.5 Fittest strategy of Testona 1

```
random-unexecuted-action-of-type type-of-action-of
    If
        number-of-actions-of-type
            type-of-action-of previous-action
        Is greater than
        number-of-actions-of-type
            type-of-action-of random-most-executed-action
    Then
        random-action-of-type type-of-action-of
            If left-clicks-available Then
                random-action
            Else
                random-unexecuted-action
            EndIf
    Else
        random-action-of-type type-of-action-of
            If state-has-not-changed Then
                random-action
            Else
                random-unexecuted-action
            EndIf
    EndIf
```

Algorithm 7.6 Fittest strategy of Testona 2

```
If drag-actions-available Then
    random-unexecuted-action-of-type hit-key-action
Else
    random-action-of-type-other-than type-action
EndIf
```

Algorithm 7.7 Fittest strategy of Testona 3

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

Algorithm 7.8 Fittest strategy of Testona 4

```
If number-of-actions Equals number-of-type-actions Then
    random-unexecuted-action-of-type drag-action
Else
    random-unexecuted-action-of-type click-action
EndIf
```

7.2 Phase B: Run strategies on all SUTs

The fittest strategies listed in Section 7.1 are used in the second phase of the experiments. Tests are run with each of these action selection strategies on all three SUTs. Statistical analysis is then done on the results of these tests to be able to answer the research questions. For some of these research questions a larger list of strategies is desirable in order to have more reliable results. Therefore the tests are not only run using the number one fittest strategy of each evolution, but also the numbers two and three. With duplicates removed this results in a list of 29 strategies from the evolutions, which are listed in Appendix A. Together with the random and the EARV strategy, shown in Algorithm 7.9, this results in a list of 31 strategies. The strategies are all run on each of the three SUTs. 20 runs are done per strategy with sequence length 1000, and 20 with sequence length 100.

Algorithm 7.9 The EARV strategy [17]

```
If number-of-left-clicks Greater Than number-of-type-actions Then
    random-action-of-type
        click-action
Else
    random-unexecuted-action
EndIf
```

First the used statistical tests are explained in Subsection 7.2.1. In Subsection 7.2.2 each of the hypotheses are accepted or rejected based on statistical analysis of the experiment data.

7.2.1 Statistical background

To test if the results of the experiments are significant, two statistical tests are used. The Mann-Whitney U-test [30] is used for pairwise comparison of two groups of observations, while the Kruskal-Wallis H-test [27] is used for comparison of more than two groups of observations. Both tests do not assume a normal distribution of the data, but use the relative ranking to compare results between groups. Both tests are explained in more detail below. Furthermore the definition of outliers is explained, as they are removed from the data before the statistical tests are applied.

Kruskal-Wallis H-test

The Kruskal-Wallis H-test is a statistical test suitable for comparing more than two groups of observations for which it cannot be assumed that the data is normally distributed. The null hypothesis of the Kruskal-Wallis H-test is that all observations in all groups are part of the same population. In that case differences between the observations of groups are so small that they could be explained by chance rather than by the groups being significantly different. In the context of this research, the observations are the fitness values found in each test run, while the groups are the strategies. The H-test null hypothesis translated to the context of this research is then that the results of the strategies are not significantly different, meaning that all considered strategies test equally well.

For the H-test, all observations across all groups are ranked by their value. When two or more observations have the same value, they are all assigned the

average of the ranks in their range. The following formula is used to calculate the value H.

$$H = (N-1)\frac{\sum_{i=1}^{g} n_i(\bar{r}_{i\cdot}-\bar{r})^2}{\sum_{i=1}^{g}\sum_{j=1}^{n_i}(r_{ij}-\bar{r})^2}$$

The symbols have the following meanings:

- n_i Number of observations in group i
- g Number of groups
- r_{ij} Rank (among all observations) of observation j in group i
- N Total number of observations of all groups
- $\bar{r}_{i\cdot}$ Average rank of the observations in group i
- $\bar{r}$ Average rank of all observations

From the H value a χ^2 value is calculated. The threshold value α for χ^2 is chosen as 0.05, as this is the usual threshold for this test. When χ^2 is smaller than α, the null hypothesis is rejected.

Mann-Whitney U-test

The Mann-Whitney U-test is a statistical test suitable for comparing two groups of observations for which a normal distribution of the data cannot be assumed [30]. Similar to the Kruskal-Wallis H-test, the null hypothesis is that the two groups of observations come from the same population. For the U tests, two groups of observations are selected and ranked by their value across the two groups. When two or more observations have the same value, as in the Kruskal-Wallis H-test, they are all assigned the average of the ranks in their range.

The following formula is used to calculate a value U:

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2}$$

$$U_2 = R_2 + \frac{n_2(n_2+1)}{2}$$

$$U = min(U_1, U_2)$$

The symbols have the following meanings:

- U_i The U-value for group i
- n_i The number of observations in group i
- R_i The sum of the ranks in group i

The formula shows that a U-value is calculated for both groups, and of these two U-values the smallest one is used. The resulting U-value is compared to a threshold value related to the group sizes, for which reference tables exist. If U is smaller than the threshold value, the difference between the two groups is significant, so the null hypothesis is rejected. Table 7.1 shows a part of the reference table as used for this research, for the α-value of 0.05. In our data set all strategies have 20 observations, but as the outliers are removed the group size can be smaller than 20. Therefore the reference table is shown for group sizes 16 to 20.

Table 7.1: The reference table for the threshold value for U, for the Mann-Whitney U-test, with $\alpha = 0.05$ [32]

	16	17	18	19	20
16	75	81	86	92	98
17	81	87	93	99	105
18	86	93	99	106	112
19	92	99	106	113	119
20	98	105	112	119	127

Outliers

Before the statistical tests are applied, outliers are removed from the data. An outlier is defined as a result outside the upper boundary UB or lower boundary LB for group i. The boundaries are defined as below, with UB_i as the upper boundary of group i, LB_i as the lower boundary of group i, UQ_i as the 75th percentile of group i and LQ_i as the 25th percentile of group i. In most cases, the groups are strategies, and the outliers are the tests for that strategy with a result that is outside the boundaries for that strategy. When the observations of multiple strategies are grouped together for a statistical test, the outliers are defined by the boundaries of the total group.

$$UB_i = UQ_i + 1.5 * (UQ_i - LQ_i)$$

$$LB_i = LQ_i - 1.5 * (UQ_i - LQ_i)$$

7.2.2 Analysis per hypothesis

The statistical tests and the fitness data of the runs are used to either accept or reject each of the hypotheses stated in Chapter 3. The fitness values of the fittest strategies are shown in Figures 7.2 to 7.4 in box-and-whisker diagrams, for each of the SUTs and each of the sequence lengths. Note that a lower fitness value is better.

Hypothesis 1: A strategy found using the complex evolution setup is better than random action selection.

To find out if this is true, the performance of the fittest strategy of each of the evolution runs is compared to the performance of the random strategy using the Mann-Whitney U-test. If a strategy resulted from an evolution run on SUT A, the performance on SUT A of that strategy and the random strategy are compared. Table 7.2 shows the results of these pair-wise comparisons for both sequence lengths. The best performing strategy, either 'Random' or 'New' (the newly evolved strategy) is shown for each comparison. If the Mann-Whitney U-test shows that the difference is not significant, 'NSD' (no significant difference) is shown.

Table 7.2 shows that in only one of the cases the strategy resulting from the evolution performs better than the random strategy. In six of the 16 cases the

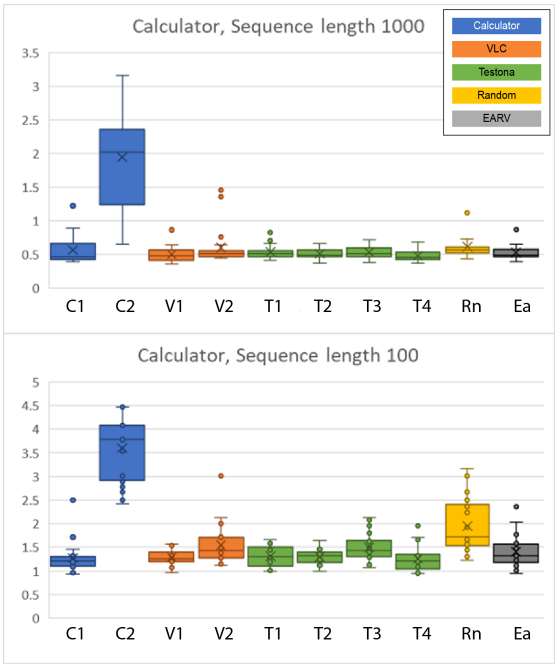

Figure 7.2: The fitness values for test runs on Calculator, per strategy. The color of the strategy indicates the SUT it has been evolved on.

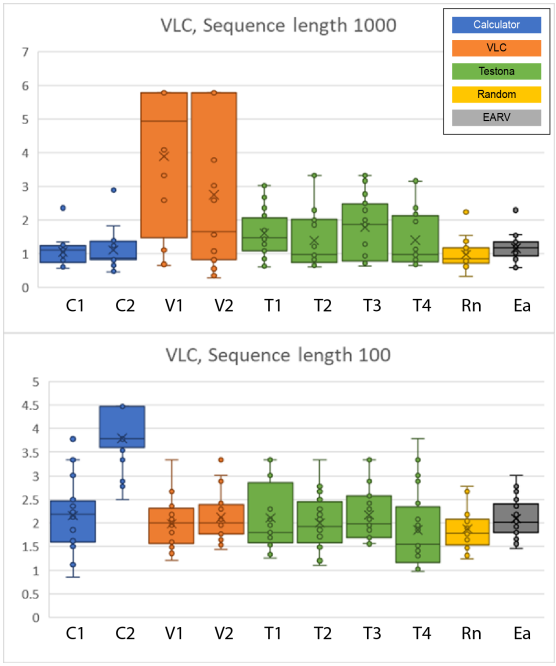

Figure 7.3: The fitness values for test runs on VLC, per strategy.

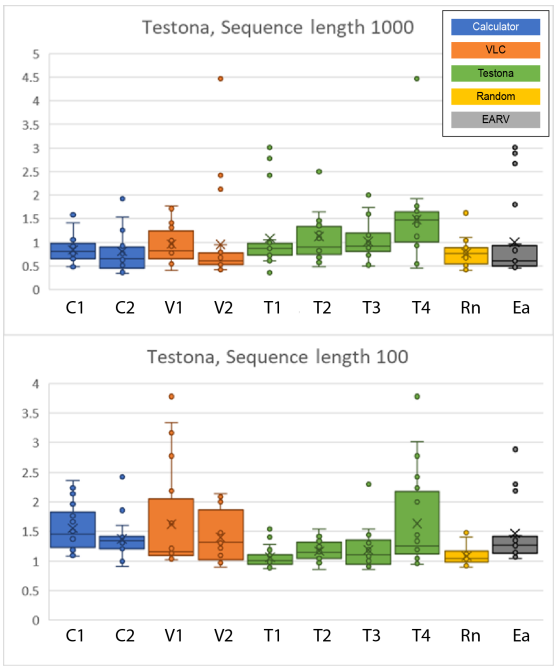

Figure 7.4: The fitness values for test runs on Testona, per strategy.

Table 7.2: Best strategy of each evolution compared with the random strategy, for sequence lengths 1000 and 100.

Strategy		**Sequence length**	
		1000	**100**
Calc	1	NSD	*New*
	2	Random	Random
VLC	1	Random	NSD
	2	Random	NSD
Testona	1	NSD	NSD
	2	Random	Random
	3	Random	NSD
	4	Random	Random

Table 7.3: Best strategy of each evolution compared with the EARV strategy, for sequence lengths 1000 and 100.

Strategy		Sequence length	
		1000	**100**
Calc	1	NSD	New
	2	EARV	EARV
VLC	1	EARV	NSD
	2	EARV	NSD
Testona	1	EARV	New
	2	EARV	NSD
	3	EARV	NSD
	4	EARV	NSD

difference in performance between the two strategies is not significant, leaving the majority of the cases in which the random strategy performs better. Hypothesis 1 is therefore rejected.

Hypothesis 2: A strategy found using the complex evolution setup is better than the EARV strategy.

The same approach as above is taken to accept or reject this hypothesis, by comparing the performance of the fittest strategy of each of the evolutions with the performance of the EARV strategy, using the Mann-Whitney U-test. The results are shown in Table 7.3 for each of the sequence lengths.

In two of the cases the newly evolved strategy performs better than the EARV strategy. In exactly half of the cases the EARV strategy performs better than the newly evolved strategy. This means that Hypothesis 2 is also rejected.

Hypothesis 3: There is a difference between SUTs in how a strategy performs.

Proving this hypothesis is more complicated than the previous ones, as the fitness values cannot be compared across SUTs. The fitness value is based on the number of visited states and the length of the longest path. The number of possible states differs between SUTs, therefore it may be easier to obtain a certain fitness value on one SUT than on another SUT.

Therefore the ranking of the observations of strategies are compared instead of their fitness values. First the ranking of each observation of a set of strategies is determined for each of the SUTs, the same way as done for the statistical tests described above. Next these rankings of each observation of one strategy across all SUTs are again ranked, so that the relative performance for the strategy on each of the SUTs can be compared. For example, the observations of a strategy may be ranking very high on one SUT, while the observations of that same strategy on a

Table 7.4: The number of strategies for which the performance is significantly different across the SUTs

	1000	**100**	**Joined**
Yes	19	12	22
No	12	19	9

different SUT may rank very low. Kruskal-Wallis is applied per strategy to see if the relative performance of that strategy is significantly different per SUT.

Because the purpose of this statistical test is not to find out whether the evolution yielded the best strategy and because having more data gives a more reliable result, all top three strategies plus the random and the EARV strategy are used in the calculation. Table 7.4 shows the number of strategies for which a significant difference in performance exists across the three SUTs, for both sequence lengths. The column 'Joined' shows the number of strategies for which a difference exists in at least one of the sequence lengths.

As the results show, more strategies show a difference in performance across SUTs in sequence length 1000 than in sequence length 100. Although not all strategies show a difference, 22 out of 31 show a difference in performance in at least one of the sequence lengths. Because this is the majority, Hypothesis 3 is accepted.

Hypothesis 4: A strategy evolved on SUT A works better on SUT A than strategies found on SUT B, and a strategy found on SUT A works better on SUT A than on SUT B.

The proof of this hypothesis is done in two steps. The first step is to prove that a strategy evolved on SUT A works better on SUT A than a strategy found on SUT B, and the second step is to prove that a strategy found on SUT A works better on SUT A than on SUT B. For both steps the fittest strategies are grouped per SUT they were evolved on.

To prove that a strategy evolved on SUT A works better on SUT A than a strategy found on SUT B, the performances of the grouped strategies are compared for each of the SUTs, using Kruskal-Wallis. The statistical analysis shows that for each of the SUTs and each of the sequence lengths the difference is significant, suggesting that there is a significant difference in how a strategy performs on a SUT based on the SUT it was evolved on. For each of the sequence lengths on each of the SUTs, Table 7.5 shows which group of strategies, based on the SUT they are evolved on, scores best.

The results show that in only one of the cases (VLC, sequence length 100, in **bold**) the strategies that are evolved on a SUT score better on that same SUT than strategies evolved on other SUTs. In all other cases, strategies of a different group score better. The first part of hypothesis 4 is therefore rejected.

To prove that a strategy found on SUT A works better on SUT A than on SUT B, a similar analysis is done as for Hypothesis 3. The number one strategies of all evolutions are analyzed on how they perform across SUTs. Per SUT, the observations of the eight strategies (two for Calculator, two for VLC and four for

Table 7.5: Performance of strategies on SUTs grouped by SUT the strategies were evolved on

	Length	Best		Worst
Calc	1000	VLC	Calc	Testona
	100	VLC	Calc	Testona
VLC	1000	Calc	VLC	Testona
	100	**VLC**	Calc	Testona
Testona	1000	Calc	VLC	Testona
	100	VLC	Calc	Testona

Testona) are ranked. The ranks of each observation of each strategy across the three SUTs are then again ranked, to be able to compare the relative scores across SUTs. Kruskal-Wallis is applied for each strategy to compare the rankings of observations across SUTs. Table 7.6 shows on which SUT each strategy scores best. If Kruskal-Wallis shows that the difference is not significant, 'NSD' (no significant difference) is shown.

In seven of the 16 cases there is no significant difference in the performance of a strategy across SUTs. In three cases, shown in **bold**, the strategy performs best on the SUT it has been evolved on. This is a minority of the cases, so both parts of Hypothesis 4 are rejected.

Main hypothesis: A better strategy is found specifically for each SUT by using the complex evolution setup.

To prove whether this is true, the sub-hypotheses and their judgments are summarized in Table 7.7. All sub-hypotheses except Hypothesis 3 have been rejected. In general the found strategies are not better than the random strategy or the EARV strategy. Although strategies appear to be system-specific, the strategies found on a SUT do not perform better on that SUT than strategies found on other SUTs, and do not perform better on that SUT than on other SUTs. The main hypothesis is therefore rejected.

Table 7.6: Performance of strategies on SUTs, grouped by the SUT the strategies were evolved on

Strategy		Length	Best		Worst
Calc	**1**	1000		NSD	
		100	**Calc**	VLC	Testona
	2	1000	Testona	VLC	Calc
		100	Testona	Calc	VLC
VLC	**1**	1000	Calc	Testona	VLC
		100		NSD	
	2	1000	Testona	Calc	VLC
		100		NSD	
Testona	**1**	1000		NSD	
		100	**Testona**	Calc	VLC
	2	1000		NSD	
		100		NSD	
	3	1000		NSD	
		100	**Testona**	VLC	Calc
	4	1000	Calc	VLC	Testona
		100	Calc	VLC	Testona

Table 7.7: Summary of sub-hypotheses

Nr	Hypothesis	Judgment
1	A strategy found using the complex evolution setup is better than random action selection.	False
2	A strategy found using the complex evolution setup is better than the EARV strategy.	False
3	There is a difference between SUTs in how a strategy performs.	True
4a	A strategy found using the proposed evolutionary computing method works better on the SUT used to evolve it than strategies found on other SUTs.	False
4b	A strategy found on one SUT works better on that SUT than on other SUTs.	False

Chapter 8

Discussion

The previous chapter has shown that the main hypothesis has been rejected. This however does not mean that the entire research can be discarded. This chapter discusses possible causes of the negative results and the gained insights that can be used in future research and in the development of TESTAR. First the results are put in context, comparing the results with those of EARV and carrying out further analysis. Next the used evolution setup, the factors that may have caused these results and the threats to validity are discussed. Finally, based on these findings a research agenda is defined to investigate further improvements on action selection in TESTAR.

8.1 Results context

The results of the presented research may seem disappointing compared to the results of EARV [17]. To put the results in the right perspective, additional analysis is carried out on the test results to answer a number of questions arising from the previous chapter. These questions are answered in the following subsections.

- Is the strategy found in EARV really better than the random strategy?
- Are there strategies among those found in this research that are better than the random or the EARV strategy?
- Does using evolutionary computing with the used setup add value in looking for a better strategy?

8.1.1 Comparison with EARV

In EARV [17] the found strategy performs better than the random strategy in several cases. As presented in the previous chapter, the strategies found in this research do not perform better than the random strategy. In order to confirm that the EARV strategy is indeed better than those found in the presented work, the performance in both researches of the random strategy and the EARV strategy is compared.

Table 8.1 shows a summary of the EARV results as presented in Chapter 2. The best performing strategy is listed, being either the random strategy (RND) or the EARV strategy, for each used metric and for each of the SUTs. $\mathcal{Q}$-learning [44],

Table 8.1: The best scoring strategy in EARV [17] per metric, comparing the EARV strategy to the random strategy (RND).

	PowerPoint	**Odoo**	**Testona**
Abstract states	EARV	EARV	RND
Longest path	EARV	EARV	RND
Max state coverage	RND	EARV	EARV
Min state coverage	EARV	RND	EARV

Table 8.2: The best scoring strategy in the presented research per metric used in EARV, comparing the EARV strategy found to the random strategy (RND)

	Calculator		**VLC**		**Testona**	
	1000	**100**	**1000**	**100**	**1000**	**100**
Abstract states	EARV	NSD	NSD	NSD	NSD	RND
Longest path	NSD	EARV	NSD	NSD	NSD	RND
Max state coverage	NSD	NSD	NSD	NSD	NSD	NSD
Min state coverage	NSD	RND	NSD	NSD	NSD	RND

which is a third approach used in the tests of EARV, has been omitted from the table, as it has not been tested in the presented work. All tests used a sequence length of 1000. The Kruskal-Wallis H-test performed in EARV shows that the differences between the strategies are significant for all metrics on all SUTs. It is easy to see that the EARV strategy performs better than the random strategy in many cases.

Table 8.2 shows the same presentation of the results found in the presented research, using the same two strategies and the same metrics. The performance of the strategies is compared pair-wise using the Mann-Whitney U-test. In most cases no significant difference (NSD) is found in the performance of the two strategies. Only on the Calculator does the EARV strategy perform better on two metrics in two different sequence lengths.

A difference between the two researches is the used SUTs. The SUT used during the evolution in EARV is PowerPoint. It is possible that the positive results are not found because PowerPoint is not used. The only SUT that is used in both researches is Testona. Since the sequence length used in the EARV tests is 1000, the column Testona in Table 8.1 represents the same test scenario as the column Testona - 1000 in Table 8.2. In the new tests no significant difference between the strategies was found for any of the metrics, while EARV did show a difference. EARV notes that the performance of the found strategy on Testona is worse than on the other two SUTs. The influence of the used SUTs on the performance of the EARV strategy cannot be excluded. Still it is noteworthy that the positive results on multiple metrics, as presented in EARV, are not reproduced in the presented research on any of the SUTs used here.

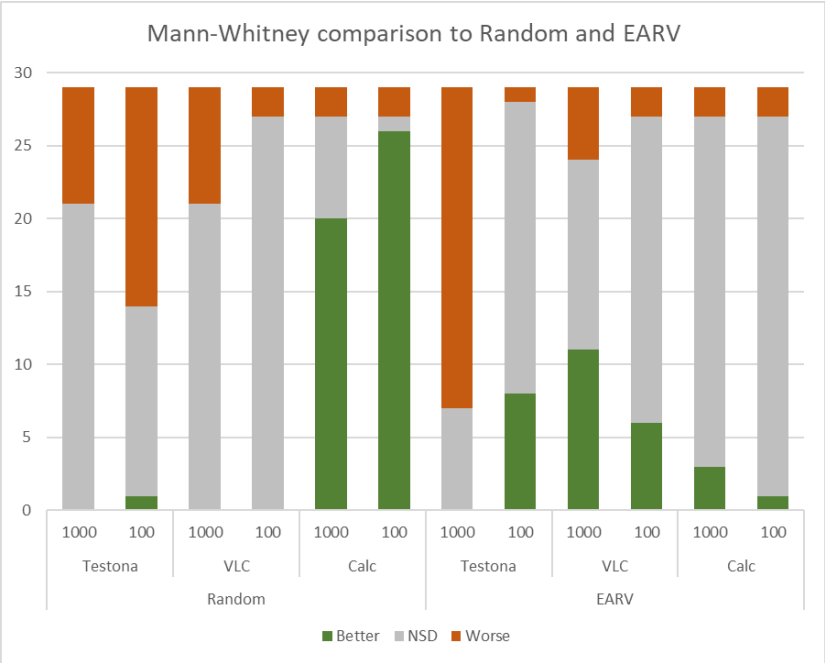

Figure 8.1: The number of the 29 strategies performing significantly better or worse, or with no significant difference (NSD) compared to the random strategy or the EARV strategy.

8.1.2 Comparison of all strategies with random and EARV

A reason that no better strategy than random action selection is found in the presented work can be that random action selection is the best possible strategy. The non-existence of better strategies cannot be proven without testing every possible strategy, but the presence of a better strategy in the list of the 29 tested strategies can be checked. Therefore all 29 strategies are compared one on one to the random strategy. These comparisons are also done with the EARV strategy. The Mann-Whitney U-test is used to see if the difference in performance is significant. The 29 found strategies are compared to both random action selection and the EARV strategy for each of the three SUTs and each of the two sequence lengths, resulting in $29 \times 2 \times 3 \times 2$ comparisons.

Figure 8.1 shows how many of the 29 strategies perform significantly better or worse than, or with no significant difference compared to the random strategy or the EARV strategy, for each SUT and each sequence length. Noteworthy is that 26 of the strategies on the list perform significantly better than the random strategy on the Calculator with sequence length 100, and 20 with sequence length 1000. However, only one of the strategies performs significantly better on Testona and none on VLC. On Testona with sequence length 100, 15 of the 29 strategies perform worse than the random strategy.

Of the 29 strategies, 22 perform significantly worse than the EARV strategy on Testona with sequence length 1000. This number is much lower in the other cases with the EARV strategy. On VLC with sequence length 1000, 11 of the 29 strategies perform significantly better.

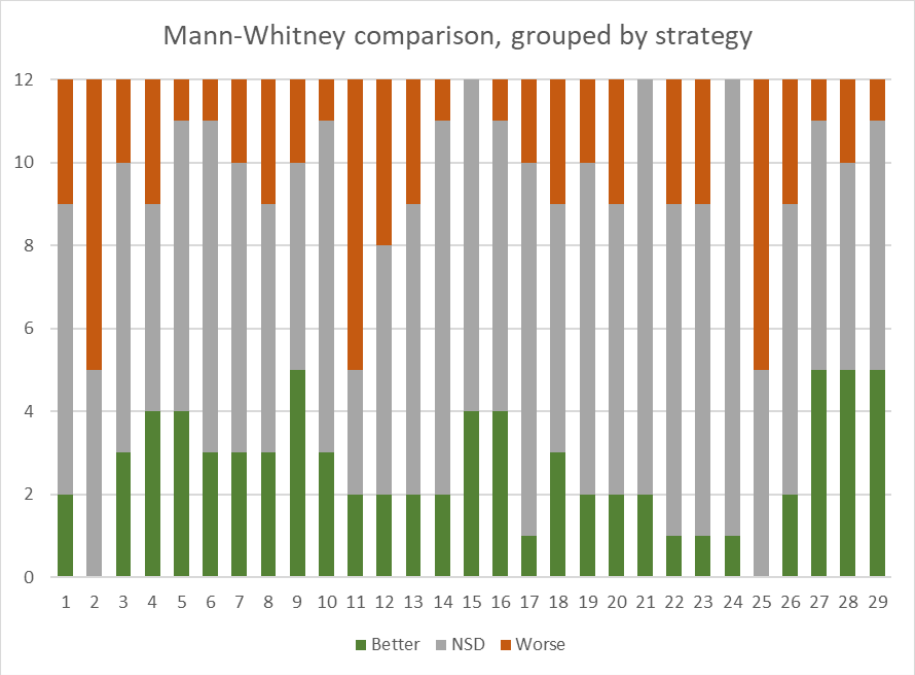

Figure 8.2: The number of cases in which each of the 29 strategies performs significantly better or worse, or with no significant difference compared to the random strategy or the EARV strategy.

The figure does not show whether the same strategies perform better or worse each time. Figure 8.2 shows the same comparisons grouped by strategy. The number of comparisons per strategy is 12, with three SUTs $\times$ two sequence lengths $\times$ two strategies to compare with, being the random and the EARV strategy. The figure shows that none of the strategies scores consistently better than either the random strategy or the EARV strategy.

The question asked at the start of this section is whether a better strategy than the random or the EARV strategy is among the 29 tested strategies. The answer to that question is not a clear yes, as none of the strategies score consistently better than the random or EARV strategy.

8.1.3 Added value of evolution

An assumption that is made when using evolutionary computing is that children of individuals have a similar fitness to their parent. Taking an individual with a good fitness value and mutating it, will then most likely result in an individual that also has a good fitness value, and hopefully better. Because the individuals with a better fitness have a higher chance of getting selected for mutation, the population resulting from these individuals should have a better fitness value than the population in the preceding generation. This principle is why evolutionary programming can help in finding a good solution to a problem. If the fitness of parents and children is not related, using evolutionary computing does not have added value over generating individuals randomly, trying them all and selecting the best one.

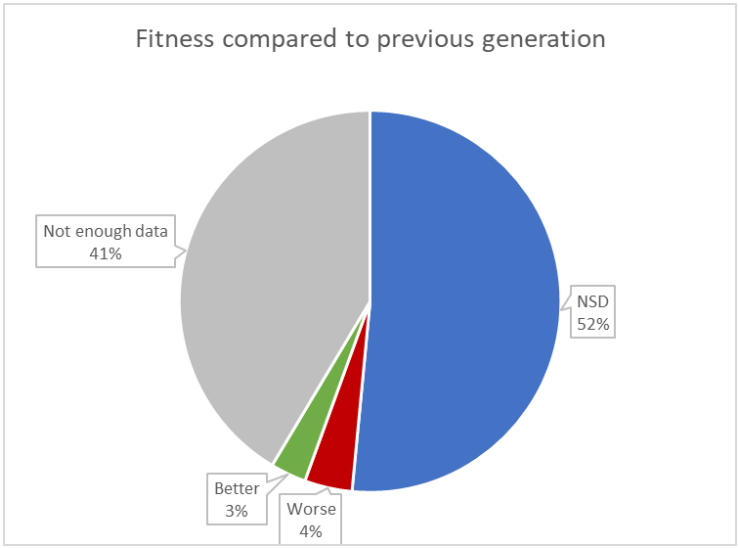

Figure 8.3: The percentage of cases in which the fitness value of the new individuals of a population is better or worse than, or with no significant difference compared to those of the previous generation.

Because ECJ does not report which individuals result from which parents, it is not possible to calculate whether fitness scores of genetically related individuals are similar. What can be observed is the fitness values of the new individuals per generation. The individuals that already occurred in previous generations are not considered, as these are most likely the elite individuals. The elite is not a result of mutations or other operations on individuals.

To find out whether the fitness value increases over generations, the Mann-Whitney U-test is used to compare the fitness values of the newly occurring, canonical, unique individuals in a generation with those of the immediately preceding generation. With an evolution of 10 generations, there are 9 comparisons of successive generations to be made. Furthermore the last generation is compared to the first, to see if there is an effect over multiple generations. With four evolutions on each of the three SUTs, this results in 120 comparisons in total.

The results of these comparisons are shown in Figure 8.3. In only 3% of the cases is the fitness values of the newly occurring individuals significantly better than those of the new individuals in the previous generation. In 4% of the cases the fitness value is worse. In 52% of the cases there is no significant difference (NSD) between the fitness values. In the rest of the cases there are fewer than four unique new individuals in one of the two compared generations, which is considered too little to measure an effect. None of the last generations show a significant difference in fitness compared to the first, randomly created generation.

These results give little reason to believe that evolving strategies from previously found fit strategies is better than randomly generating the strategies. Using evolutionary computing may not add value in solving the problem of finding a better action selection strategy, at least not in the current setup.

8.2 Experiment setup

As the previous section shows, the current evolution setup does not lead to the desired results. This section analyzes the influence of the experiment setup on the found results. The used evolution setup is reconsidered, looking for possible causes and opportunities for improvement. First the complexity of the strategy trees and their components are discussed, then the used parameter values for the evolution. The last subsection discusses threats to validity based on these and other factors.

8.2.1 Complexity of the strategy trees

One of the main differences between EARV and the presented research is the complexity of the strategy trees. The purpose of the presented research as described in Chapter 3 is to find a better action selection strategy using a more complex tree representation, with more degrees of freedom. Since a better strategy is not found using this complex tree representation, the question arises how this can be explained.

A more complex tree representation with more degrees of freedom allows for more different strategies. The assumption at the start of the research was that this increases the chance of having the best possible strategy among them. Then why is this best strategy not found in the tests?

The large number of possible strategies with the complex tree representation not only increases the potential number of good strategies, but also the total number of possible strategies. The evolution starts with randomly generating the configured number of individuals from the given components to create the first generation. The configured number of individuals as explained in Chapter 5 is 100. In each following generation the same number of individuals is created by operations on selected individuals. As there are 10 generations, in theory $10 \times 100 = 1000$ strategies are tested in each evolution run.

In practice, these numbers are much lower, because of elitism, duplicates and logical equivalence. The strategies in all following generations depend on the strategies in the first generation, which increases the chance on duplicates and logical equivalence. The evolution runs that are done for this research have between 200 and 250 unique, canonical strategies over 10 generations. A quick test that randomly generates as many strategies as possible yields more than 120,000 unique, canonical strategies. Assuming that only a small number of strategies are better than the random strategy, the chance of having such a strategy among a nearly random selection of 250 out of more than 120,000 is small.

EARV uses fewer degrees of freedom, with an if-statement as the fixed root and a limited set of components. Although Subsection 8.1.1 shows that the EARV strategy does not always perform better than the random strategy, it may be better to have fewer degrees of freedom. That way the results can be more controlled by an intelligent design of components.

8.2.2 Evolution parameters

This subsection analyzes the parameter settings of the evolution, their possible influence on the results and the potential for further research. For each of the evolution phases, being creation, evaluation, selection and operation, the chosen parameter values are discussed below.

Creation
If the number of randomly generated individuals in the creation phase is increased, more different strategies will be tested, increasing the chance of finding the best possible strategy. Similarly, more generations can be evolved, again allowing more strategies to be tested. However, because the evolution can already take up to a week with the current settings, significantly increasing these values will take a significant amount of extra time. This is not considered a viable solution.

Evaluation
When evaluating the individual, relevant settings are the number of test runs, their sequence length and the fitness function. As with the number of individuals and generations, increasing the number of test runs and their sequence length will increase the time needed for the evolution. The optimization of these parameters as done for this research is described in Chapter 5. Not much gain is expected by increasing these numbers.

The fitness function currently depends on the number of visited states and the length of the longest path. The two numbers are simply added up, while the number of visited states is always larger than or equal to the length of the longest path. As the evolution optimizes for the chosen fitness function, the evolution results alone cannot tell whether this is a good basis. The results are evaluated with the same metrics as the individuals during the evolution, so no conclusions on the validity of the metrics can be drawn without a broader perspective on the metrics. More research is needed on this topic, which is discussed in the next section.

Selection
In the selection phase, both the elite selection and tournament selection are used. The elite comprises the ten best individuals of the generation. In the first generation with 100 unique individuals, the elite will comprise ten unique individuals. It must be noted that some of these may be logically equivalent and therefore are given the same fitness value. In later generations, the number of unique, canonical strategies is much lower than 100, while the population size is still 100. This increases the chance of having duplicates in the elite selection, which reduces the diversity of the population. Although it seems tempting to take the best ten individuals to the next generation, this has a negative effect on the diversity of the population. This effect was not measured in the analysis done to optimize this parameter in Chapter 5, as the used number of individuals per generation during the optimization tests was 1000 instead of 100.

The tournament selection randomly selects a configured number of individuals from the population, allowing the same individual to be selected multiple times. The chosen value is 15 individuals. From these individuals, the one with the best fitness value is selected and put in the operation pipeline to create a new individual. This is repeated until the desired population size is reached. In a population of 100 individuals, the chance of selecting an individual that is also part of the elite of size 10 is 80%[1], every time the tournament is repeated. Although these individuals are mutated after selection, selecting the same individuals each time has a negative effect on the diversity of the new population. Soon all individuals may be brothers, although still slightly different. It may therefore be better to decrease the tourna-

[1]Calculated as follows: $(1 - (\frac{9}{10})^{15}) \times 100\%$

ment size. With a tournament size of five the chance of selecting an elite member is only 40%.

Operation
In the operation phase the selected individuals are mutated to create new individuals, using one of the operations of reproduction, cross-over, mutation, mutate-one and mutate-all. In the argumentation in Chapter 5 for the combination of these five operations, the tests only measure the diversity resulting from each of the operations, and from the combination of the five. While diversity results in more different strategies being tested, there may be other effects to take into account. Subsection 8.1.3 shows that in the current setup the fitness of the individuals does not increase over generations. This may be due to the configured operations. It is possible that there is a stronger correlation in fitness between a parent and its child resulting from cross-over, than between a parent and its child resulting from mutation. Elaborate tests are needed to measure these differences between the operations.

To summarize, possibilities for further research focusing on the evolution setup are seen in the fitness function, selection and operation parameters. Possible future research is further explained in Section 8.3.

8.2.3 Threats to validity

Cook *et al.* [11] distinguish four types of (threats to) validity: conclusion, internal, construct and external validity. Each of these types is discussed below.

Conclusion validity
Conclusion validity concerns the relationship between the results and the conclusions. To have a more solid base for the conclusions that are drawn, each strategy could have been evaluated with 30 or more instead of 20 runs, or with longer sequences. This would make evolutions take significantly more time. The performance of strategies varies greatly between runs. The root of this problem lies in the large factor of randomness due to the chosen components. Solutions are suggested in Section 8.3.

Internal validity
Internal validity concerns whether the relationship between the experiments and the results is causal, and not influenced by other factors. A threat to internal validity is that a SUT's behavior when starting is influenced by how it was shut down last time. Testona remembers the windows and files that are open when the program is shut down, and shows them when restarted. This may influence the metrics of the next runs. In VLC and the Calculator the previous run has little influence on how the application starts, except for a few settings. It would be better for validity of results to start the application each time with the exact same windows and settings. This is not possible for some applications.

Another potential threat to internal validity is that the experiments were run in five different virtual machines (VM). These were originally copies of the same VM, but the experiments may have caused changes to the VMs. The effect on the results, if any, is not expected to be significant.

Construct validity
Construct validity focuses on the theory behind the experiments and how it relates to the reality of the experiments. As discussed earlier, the used fitness function may not be a good measure for how well a SUT is tested. More research is needed to determine which metrics are a reliable input for the fitness function, and what weight should be given to each of them.

External validity
External validity concerns to what extent the results can be extrapolated to other situations. This research is limited to three desktop applications as SUTs. The results may be different on web applications or other types of applications.

8.3 Research agenda

Several suggestions are made for further research, making it important to consider where to start. As no valid conclusions can be drawn if the metrics are not right, the first aspect to address is the metrics used to distinguish good strategies from bad ones. Evolution takes time, so a metric that can predict the performance of a strategy in long runs from its performance in short runs can save a significant amount of time.

As soon as the metrics to be used as input for the fitness function are clear, the evolution setup can be further improved. The set of components that strategy trees can consist of, can be improved by trying and evaluating new components and/or eliminating existing ones. The selection parameters can be changed to see what the effect is on the results. The operation parameters can be further analyzed to see if there is a difference in fitness between the children they produce from fit parents.

A next step can be to create a whole new setup to explore the possibilities of using other forms of machine learning. A larger, long-term research project can focus on this topic. Each of the research topics is more elaborately described in the following subsections.

8.3.1 Metrics

To be able to draw any valid conclusions on whether a strategy works well, reliable metrics are needed. More research is needed to determine whether there is a correlation between 'testing well' and the two used metrics, number of visited states and length of the longest path. Other metrics should be considered as input for the fitness function. Evolution requires many runs to evaluate individuals, so the runs need to be short. The short runs are not suitable to take the number of encountered faults or the percentage of code coverage as reliable metrics. Both numbers are likely too small to base conclusions on.

Further research can use long test runs and investigate the correlation of these metrics with metrics that are easier to measure in short runs. For example, if long runs in which many faults are detected also measure a higher number of visited states, this would show that the number of visited states is a valid input for the fitness function. The number of visited abstract states may be an even better metric, depending on the definition of an abstract state. Follow-up research can assert which control parameters should be ignored for the optimal abstract state definition for this purpose.

8.3.2 Strategy tree components

With the current components, actions are selected based on their action type and whether they have been executed before. Within such a set, for example all unexecuted type actions, a random action is selected. This leaves a large component of randomness. The boolean components base their value on factors such as the number of available type actions or whether the state has changed after the most recent action.

There are more parameters that could be taken into account, for example the absolute or relative position of a button, the group it belongs to or its color or size. Composite components can be created, consisting of multiple components from the current set. Instead of giving ECJ the freedom to create nonsensical boolean statements such as 'number of type actions equals random number', a set of useful, predefined boolean statements can be created, such as 'number of click actions is greater than number of type actions'.

Other possible components could select multiple actions in a sequence. A simple component could be a 'first ... then ...' node, for example 'first select a random click action, then select a random type action'. Instead of returning a single action, the strategy can return an array of actions. As long as there are actions in the array, TESTAR will executed these. When the array is empty, TESTAR uses the strategy again to select a list of actions.

When a form is encountered in the SUT, an approach can be to first fill all text fields of the form and then click a button. TESTAR could be configured to recognize a form as such, by observing a group of components with a set of text fields and a button. New strategy components could be the action 'fill out form' and the boolean 'there is a form'. More of such composite actions can be thought of to create more of these types of components.

Inspiration for new components can also be drawn from algorithms such as $\mathcal{Q}$-learning [44]. A component can take the $\mathcal{Q}$-value of an action into account. The straightforward approach would be to have the component select the action with the highest value as in normal $\mathcal{Q}$-learning. A variation can be to let the probability of an action to be selected depend on its $\mathcal{Q}$-value.

Future work can investigate the effect of changing the used set of components. The set of components used in the presented work can be reevaluated, removing components that are unlikely to contribute to fit strategies, such as 'random number'. More types of composite components, as the ones described above, can be added to the set. The effect of varying the set of components can be measured by running multiple evolutions with each set and comparing the results.

8.3.3 Evolution parameters

As described in the previous section the configured evolution parameters leave room for improvement. Possibilities for further research are mainly seen in the selection and operation parameters.

As for the selection parameters, more research is needed to measure the influence of the size of the elite and the tournament selection on the fitness of the resulting individuals. A smaller elite and tournament size will decrease the chance of the fittest individuals to be selected, but this may have a positive effect on the diversity.

For the operation parameters, future work can focus on the effect of the different operations on the fitness of the next generation. Several evolutions need to be run

on different SUTs, with different operation configurations. The fitness of the new individuals in each generation then needs to be compared to the fitness of the individuals in the immediately preceding generation, as is done in Subsection 8.1.3. The results for each of the operations should then be compared, to see if any of the operations has a larger chance of yielding fit individuals when given fit individuals as input.

8.3.4 Machine learning

Evolutionary programming can be considered a machine learning mechanism. In machine learning [10] a set of parameters is learned by trying values and adjusting them, while in evolutionary programming the best individuals are mutated, hoping to get an even better individual. Other machine learning approaches can be tried to see if these will improve how well TESTAR tests. While an algorithm such as $\mathcal{Q}$-learning, which has already been tested with TESTAR [16], learns a reward value with every action it takes, machine learning could also be applied to learn what a good strategy is that is used for all action selection steps.

In the current setup the metrics that are returned concern the performance of the strategy as a whole. This is inherent in the current evolution setup with ECJ, where an individual has a single fitness value. It could be that some of the branches of the strategy are never reached, for example because a boolean value is always true in practice. When a strategy has a good fitness, this could be due to a branch that is always used and that selects an unexecuted action. When the strategy is then selected and mutated, there is a chance that this good branch is entirely replaced with a different branch or that the boolean leading to this branch is replaced so that the branch is no longer reached. Another possibility is that the other branch that was not used at all is replaced, which has no effect on the performance of the strategy.

What if a machine learning algorithm is used to analyze the branches of the trees and control the mutations? The fitness value may be attributed to a single branch of the tree, while another branch of the tree is dead. An algorithm could assign

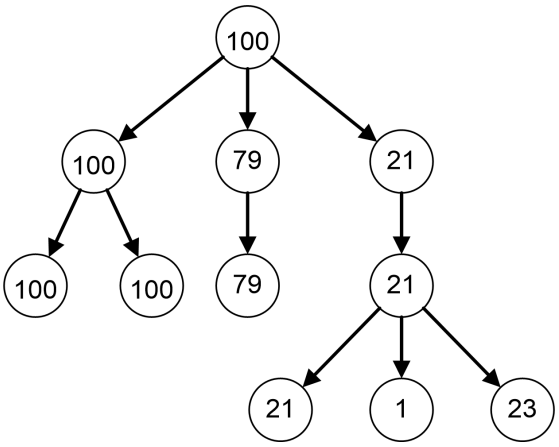

Figure 8.4: An example of a possible counter tree that shows how often each node is used during a test with sequence length 100.

values to components of the tree instead of the tree as a whole, and thus refine a strategy in a more controlled way. The algorithm could analyze the similarities between trees of strategies that work well and combine the good components. If it sees that a boolean always has the same value, it can conclude that the boolean statement is not useful and it can assign the found fitness value entirely to the used branch, while leaving the fitness value of the unused branch as unknown.

To implement this, TESTAR would need to be adapted to return information on the frequency with which nodes of the strategy are used when selecting actions. A simple counter in each object representing such a node could be incremented each time the getValue method is called, and a string of counter values can be returned to the machine learning engine, similar to the string representation of the strategy trees. The machine learning engine can map these values on the components of the strategy. The information can be combined with the other metrics that are returned by TESTAR, allowing the machine learning engine to learn the added value of each component.

Future work can investigate how to set up such a controlled machine learning algorithm. Most likely the entire setup has to be created manually, as standard engines such as ECJ cannot handle fitness values per node and controlled mutations.

Chapter 9

Conclusions and future work

9.1 Conclusions

The purpose of this research is to find better action selection strategies for monkey testing using evolutionary computing. A better action selection strategy than randomly selecting actions can help improve black box monkey testing with TESTAR.

In order to find better action selection strategies using evolutionary computing, an evolution setup is created, using ECJ [12] to generate and evolve individuals consisting of trees that represent strategies. Components are defined that these trees can be composed from. Evolutions are run on three Systems Under Test (SUTs): Windows 7 Calculator, VLC [41] and Testona [40]. On each of these SUTs four evolutions are run, with each evolution consisting of ten generations with 100 individuals each. Each individual is evaluated by sending the strategy represented by its tree to TESTAR and running tests with it. 20 tests are run with TESTAR for each of the strategies, with a sequence length of 100 actions. Metrics of each of the runs are returned to ECJ, that then calculates the fitness of the individual based on the number of states that are visited and the length of the longest path in the generated state transition graph. Fit individuals have an evolutionary advantage with a larger chance to procreate, which should result in fitter children in the next generation. The top three best individuals at the end of each evolution are selected for further analysis.

To be able to draw conclusions on the performance of these strategies, all strategies resulting from the evolutions on any of the three SUTs are run on all of the SUTs. The selected individuals of the evolution runs are gathered on a list, resulting in 29 unique, canonical strategies. The default, random strategy and the strategy found in the previous research [17] (labeled EARV) are added to the list. Every strategy is then tested on each of the SUTs, with 20 test runs for both sequence lengths 100 and 1000. The fitness value of each of the runs, together with all the other metrics returned by TESTAR, are used for further analysis.

The best strategies from the evolution runs do not perform better than the random strategy or than the EARV strategy. A significant difference is found between how different strategies perform on different SUTs. This suggests that each SUT has a different ideal strategy. The results however show that this ideal strategy is not found using the chosen evolution setup. Strategies resulting from the evolution on one SUT do not necessarily perform better on that SUT than on other SUTs,

or than strategies evolved on other SUTs. The main research hypothesis, stating that a better strategy is found specifically for a SUT by using the chosen complex evolution setup, is rejected.

The results of the random strategy and the EARV strategy are compared to the results of EARV, in order to find out whether the EARV strategy is indeed better than the strategies found here. In the presented work the EARV strategy shows not to perform significantly better than the random strategy on any of the SUTs. Further analysis of the test results also shows that there is no reason to believe that the current evolutionary computing setup works better than randomly generating strategies. As the number of possible strategies with the current set of components and parameter values is at least 120.000 and the number of unique, canonical strategies occurring in an evolution run is around 250, the chances of finding a good strategy using the current setup are small. Furthermore the evolution runs, in which 20 tests are run with each of the strategies with 100 actions, can take up to a week to evaluate all of these 250 strategies.

To conclude, the presented work does not yield the best action selection strategy, but it does give insight into mechanisms of evolution and action selection that do or do not work. This results in a research agenda for future work on action selection strategies in traversal-based black box monkey testing.

9.2 Future work

Although the evolutions with the current setup take a long time and do not yield the desired results, the research has lead to a great range of insights that can contribute to improvements of action selection or that require further research. A research agenda is proposed to outline the future work on the subject of action selection in monkey testing. The first item on this agenda is the metrics for testing well. More research is needed to know which metrics are a good predictor of a fit action selection strategy while testing only short runs. These metrics are needed to be able to draw any valid conclusions from any work that measures the quality of action selection strategies.

When the metrics have been determined, the evolution setup can be further refined to see if this yields better strategies. The set of components that the strategies are built up of can be altered, for example by adding components that select a series of actions or that take other parameters of the actions into account than current components do. The effect of altering the selection parameters can be observed by running several evolutions with different selection parameter values. Furthermore the operations done on selected individuals, such as mutations and cross-over, can be further analyzed to see if they differ in the effect they have on the fitness of the following generations.

While the above suggestions stay close to the path of evolutionary computing, the final suggested research topic further explores the possibilities of other machine learning techniques in finding better action selection strategies. Evolutionary computing considers the fitness of an individual as a whole, but machine learning can be used to consider the fitness of parts of the tree that the individual consists of. The machine learning algorithm can combine elements that are considered good to create better strategy trees.

The proposed future work will bring black box monkey testing towards maturity, as the monkeys are getting smarter. Because monkey testing requires little prepara-

tion and maintenance, there is a low threshold for companies to start applying the method. As it matures and research shows that it adds significant value, it can be applied on a large scale. This will significantly contribute to the field of software testing, and therefore contribute to better software.

Chapter 10

Reflection

When the main hypothesis is rejected, it is easy to be discouraged by this. On the contrary, I would claim that this is what made the project challenging and interesting.

I started my search for a graduation project talking to several people at the Open University. Many of the topics proposed by teachers were already narrowed down. When I first talked to Tanja Vos about TESTAR, what appealed to me was the room for my own input and creativity, as the topic was not yet clearly defined. This also brought the challenge to find the right topic related to TESTAR.

The initial thought was to use the state transition diagram learned by TESTAR to do model-based testing. I soon learned that this was not a suitable choice, as the diagram that TESTAR creates from the states and actions it encounters is always incomplete. The next idea was $\mathcal{Q}$-learning, as there had already been a research with TESTAR on this topic. However, the results of this research were not very promising, and $\mathcal{Q}$-learning did not leave much space for creativity. Another research, although very preliminary, had more promising results, and used evolutionary computing to evolve action selection strategies. The research setup was quite basic, so there was enough room for further exploration on this topic. Evolutionary computing was a new field for me, which made it even more interesting. Little was known in literature on the application of evolutionary computing on action selection in monkey testing, leaving room for some pioneering research.

Both TESTAR and ECJ were new to me, so it took some time to get to know them and get them running. Setting up ECJ to generate strategies from components was relatively easy, as ECJ is set up for people to create individuals from their own components. Getting TESTAR to run and use the strategy was a bit more challenging, because adapting TESTAR required a deeper dive into the code.

As I did a few test runs I discovered several points that needed automation, adaptation and polishing, to make the evolutions run smoothly. Because the evolutions turned out to take several days to a week, I needed insight into the progress, so I created a panel that displayed the status. As the name monkey testing suggests, the monkeys did not behave properly and kept breaking the run environment, deleting or moving files it needed to run, by clicking around in file browser windows. This interrupted the evolutions, which made it even more challenging to have a complete evolution run. To address that problem, a pause button was introduced on the panel, and I created the functionality to load an archive of previous results.

Gradually it started to grow and become an application, including configuration files and a GUI.

The evolution runs still took a long time and had to be redone several times because of interruptions. Using five virtual machines in parallel, reusing previous results with the archives and with some persistence I finally managed to complete four evolution runs on three different SUTs. After that it was relatively easy to gather the top three strategies and run more tests with this list. I had already created the necessary components in earlier stages, and the process could be easily restarted halfway, unlike the evolution process.

I then had a large data set, so it was time to analyze it in order to draw conclusions. The statistical analysis was a nice puzzle, but the results raised more questions than that they answered. Is the analysis correct? What does this prove? I discovered that the subquestions to the main research question that I wrote down before the start of the research were not helpful. The initial subquestions were more about how to set up the evolution than about necessary findings to answer the main question. I decided to change the subquestions into the questions that I really needed. These new subquestions seemed very obvious at the end of the research, but in hindsight everything always seems clearer. Formulating the new subquestions really helped in taking the next step that was needed to come to the final conclusions.

While doing the experiments I already had several ideas on how to improve the setup, but I knew that time was limited for this research. I needed to stick to the setup I had in order to get results, which was sometimes frustrating. Instead of applying the changes directly, I gathered the ideas into the presented research agenda. The added value of this research may not have been the found action selection strategies, but the gained knowledge of how to come to a good strategy, and which approaches do and which do not work. I am confident that the future research on the research agenda will add significant value to the field of action selection strategies in black box monkey testing.

To conclude, I am glad I chose this topic for my graduation project, as it allowed me to be creative in finding solutions. It challenged me to explore several topics I was not yet familiar with. I had to apply various skills gathered in the Master's curriculum, such as software architecture and testing. A more predefined topic might have given me a higher chance on positive results, but I prefer pioneering over certainty. If I would have to do it again, I would still have chosen such a topic. However, I would have run tests and statistical analyses earlier in the process to verify the results. For example the attempt to reproduce the results of EARV could have been done much earlier.

I would like to thank my two mentors, Jeroen Keiren and Tanja Vos, for their seemingly endless patience with me, their challenging questions, their positivity and their good advice. It was a pleasure to work with them on this project. A Master thesis can be a lonely exercise, but they made it feel like a team effort.

Bibliography

[1] Pekka Aho, Emil Alégroth, Rafael A.P. Oliveira, and Tanja E.J. Vos. Evolution of automated testing of software systems through graphical user interface. In *The First International Conference on Advances in Computation, Communications and Services (ACCSE)*, pages 16–21. IARIA, 2016.

[2] Pekka Aho, Tomi Raty, and Nadja Menz. Dynamic reverse engineering of GUI models for testing. In *Control, Decision and Information Technologies (CoDIT), 2013 International Conference on*, pages 441–447. IEEE, 2013.

[3] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, 73:66–80, 2016.

[4] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual GUI testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, 20(3):694–744, 2015.

[5] Emil Alégroth, Michel Nass, and Helena H. Olsson. JAutomate: A tool for system- and acceptance-test automation. In *Software Testing, Verification and Validation (ICST), 2013 IEEE International Conference on*, pages 439–446. IEEE, 2013.

[6] Appium. `http://appium.io/`, 2018.

[7] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical User Interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694, 2013.

[8] Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. A metaheuristic approach to test sequence generation for applications with a GUI. In *International Symposium on Search Based Software Engineering*, pages 173–187. Springer, 2011.

[9] Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Claudia Figueiredo Pereira Emer. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74:45–54, 2016.

[10] Jason Brownlee. *Master Machine Learning Algorithms - Discover how they work and implement them from scratch*. Self-published, 1.8 edition, 2016.

[11] Thomas D. Cook and D. T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[12] ECJ. https://cs.gmu.edu/~eclab/projects/ecj/, 2017.

[13] EggPlant. https://www.testplant.com/resources/eggplant-ai/, 2018.

[14] Agoston E. Eiben, James E. Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[15] Omar El Ariss, Dianxiang Xu, Santosh Dandey, Brad Vender, Phil McClean, and Brian Slator. A systematic capture and replay strategy for testing complex GUI based java applications. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1038–1043. IEEE, 2010.

[16] Anna I. Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and Tanja E.J. Vos. Q-learning strategies for action selection in the TESTAR automated testing tool. In *Proceedings of META 2016 6th International Conference on Metaheuristics and Nature Inspired computing*, pages 174–180, 2016.

[17] Anna I. Esparcia-Alcázar, Francisco Almenar, Urko Rueda, and Tanja E.J. Vos. Evolving rules for action selection in automated testing via genetic programming-a first approach. In *European Conference on the Applications of Evolutionary Computation*, pages 82–95. Springer, 2017.

[18] EyeAutomate. http://eyeautomate.com/, 2018.

[19] FitNesse. http://fitnesse.org/, 2018.

[20] Hans-Gerhard Gross and Arjan Seesing. *A Genetic Programming Approach to Automated Test Generation for Object Oriented Software*. Citeseer, 2012.

[21] GUITAR. https://sourceforge.net/projects/guitar/, 2017.

[22] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do record/replay tests of web applications break? In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 180–190. IEEE, 2016.

[23] JaCoCo. http://www.jacoco.org/, 2017.

[24] JUnit. http://junit.org/junit5/, 2018.

[25] John R. Koza homepage. http://www.genetic-programming.com/johnkoza.html, 2017.

[26] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[27] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[28] Adi Livnat and Christos Papadimitriou. Sex as an algorithm: the theory of evolution under the lens of computation. *Communications of the ACM*, 59(11):84–93, 2016.

[29] Antonio Martin Lopez-Asunsolo. Master thesis of antonio martin lopez-asunsolo, universidad politecnica de valencia, 2017.

[30] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[31] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes*, 26(5):256–267, 2001.

[32] Mann-Whitney Table. `http://www.real-statistics.com/statistics-tables/mann-whitney-table/`, 2018.

[33] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[34] Odoo. `https://www.odoo.com/`, 2018.

[35] PonyGP. `http://groups.csail.mit.edu/EVO-DesignOpt/PonyGP/out/index.html`, 2017.

[36] Urko Rueda, Anna I. Esparcia-Alcázar, and Tanja E.J. Vos. Visualization of automated test results obtained by the TESTAR tool. -, 2016.

[37] Urko Rueda, Tanja E.J. Vos, Francisco Almenar, M.O. Martínez, and A.I. Esparcia-Alcázar. TESTAR: from academic prototype towards an industry-ready tool for automated testing at the user interface level. *Actas de las XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2015)*, pages 236–245, 2015.

[38] Selenium. `http://www.seleniumhq.org/`, 2017.

[39] TESTOMAT. `https://www.testomatproject.eu/`, 2018.

[40] Testona. `https://www.assystem-germany.com/en/products/testona-testdesign/`, 2017.

[41] VLC Media Player. `https://www.videolan.org/vlc/`, 2017.

[42] Tanja E.J. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar: tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[43] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932. ACM, 2006.

[44] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[45] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschögl. A framework for monkey GUI testing. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 416–423. IEEE, 2016.

[46] James A. Whittaker. What is software testing? And why is it so hard? *IEEE software*, 17(1):70–79, 2000.

[47] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

Glossary

action selection An automated method of selecting the next action that will be performed on a SUT

API Application Programming Interface

black box A system of which no information is known on what happens within the system, only what is visible on the outside

capture-and-replay testing Testing a GUI by capturing a series of actions on the GUI and replaying (reexecuting) the series later

evolutionary computing A computing technique that is based on evolution as observed in nature, with generations of individuals, selection based on both strength and luck, and mutations of the selected individuals

genetic programming A sub-area of evolutionary computing, in which the components of the evolved individuals are part of a computer program

GUI Graphical User Interface

monkey testing Testing an application without knowledge of the purpose of the actions performed

NSD No significant difference

regression testing Verifies that a SUT still functions the same after changes have been made

strategy A method or algorithm for action selection

SUT system under test

traversal-based testing Testing a GUI by automatically scraping it for possible actions and executing these

Appendix A

Strategies

All strategies that are used in the analyses in Chapter 7 are listed below, grouped by system under test (SUT) used in the evolution the strategy resulted from. The listed strategies are the fittest three of each of the evolution runs.

A.1 Calculator

Algorithm A.1

```
random-action-of-type-other-than
    type-of-action-of
        If type-actions-available Then
            random-most-executed-action
        Else
            random-action
        EndIf
```

Algorithm A.2

```
random-unexecuted-action-of-type
    type-of-action-of
        previous-action
```

Algorithm A.3

```
random-action-of-type
    type-of-action-of
        If state-has-not-changed Then
            random-action
        Else
            random-most-executed-action
        EndIf
```

Algorithm A.4

```
If
    type-of-action-of
        random-action-of-type-other-than hit-key-action
    EqualsType
    type-of-action-of
        If Not number-of-left-clicks Equals number-of-actions Then
            If
                number-of-previous-actions
                Equals
                number-of-type-actions
            Then
                random-action-of-type-other-than hit-key-action
            Else
                random-action-of-type type-action
            EndIf
        Else
            random-unexecuted-action-of-type
                type-of-action-of random-most-executed-action
        EndIf
Then
    If
        number-of-actions-of-type click-action
        Equals
        number-of-actions-of-type type-of-action-of previous-action
    Then
        random-action-of-type
            type-of-action-of
                If state-has-not-changed Then
                    previous-action
                Else
                    random-least-executed-action
                EndIf
    Else
        random-unexecuted-action-of-type type-of-action-of
            If type-actions-available Then
                random-action-of-type
                    type-of-action-of random-least-executed-action
            Else
                random-action-of-type-other-than hit-key-action
            EndIf
    EndIf
Else
    random-unexecuted-action-of-type
        type-of-action-of random-most-executed-action
EndIf
```

Algorithm A.5

```
random-unexecuted-action-of-type type-of-action-of
    If type-actions-available Then
        previous-action
    Else
        random-action
    EndIf
```

Algorithm A.6

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

Algorithm A.7

```
random-unexecuted-action-of-type type-of-action-of
    If
        number-of-unexecuted-left-clicks
        GreaterThan
        random-number
    Then
        random-most-executed-action
    Else
        If type-actions-available Then
            random-most-executed-action
        Else
            random-unexecuted-action-of-type type-action
        EndIf
    EndIf
```

Algorithm A.8

```
random-action-of-type type-of-action-of
    If type-actions-available Then
        random-unexecuted-action-of-type click-action
    Else
        random-action-of-type click-action
    EndIf
```

Algorithm A.9

```
If number-of-type-actions Equals random-number Then
    random-least-executed-action
Else
    random-unexecuted-action-of-type click-action
EndIf
```

A.2 VLC

Algorithm A.10

random-least-executed-action

Algorithm A.11

random-unexecuted-action-of-type
    type-of-action-of
        random-action-of-type-other-than
            hit-key-action

Algorithm A.12

random-unexecuted-action-of-type
    type-of-action-of
        random-action-of-type-other-than
            type-action

Algorithm A.13

random-unexecuted-action-of-type
    type-of-action-of
        random-least-executed-action

Algorithm A.14

If state-has-not-changed **Then**
    random-most-executed-action
Else
    random-least-executed-action
EndIf

Algorithm A.15

If number-of-drag-actions **Equals** number-of-previous-actions **Then**
    random-most-executed-action
Else
    random-unexecuted-action
EndIf

Algorithm A.16

```
random-unexecuted-action-of-type type-of-action-of
    If
        type-actions-available
        Or
        number-of-unexecuted-drag-actions Equals number-of-left-clicks
    Then
        random-unexecuted-action-of-type type-of-action-of
            If state-has-not-changed Then
                random-action
            Else
                random-unexecuted-action
            EndIf
    Else
        If
            number-of-actions-of-type type-action
            Equals
            number-of-actions-of-type drag-action
        Then
            If left-clicks-available Then
                random-action-of-type-other-than type-action
            Else
                random-action-of-type click-action
            EndIf
        Else
            random-action-of-type-other-than
                type-of-action-of previous-action
        EndIf
    EndIf
```

A.3 Testona

Algorithm A.17

```
random-unexecuted-action-of-type
    type-of-action-of random-most-executed-action
```

Algorithm A.18

```
Random-unexecuted-action-of-type type-of-action-of
    If
        number-of-actions-of-type
            type-of-action-of previous-action
        Is greater than
        number-of-actions-of-type
            type-of-action-of random-most-executed-action
    Then
        random-action-of-type type-of-action-of
            If left-clicks-available Then
                random-action
            Else
                random-unexecuted-action
            EndIf
    Else
        random-action-of-type type-of-action-of
            If state-has-not-changed Then
                random-action
            Else
                random-unexecuted-action
            EndIf
    EndIf
```

Algorithm A.19

```
random-unexecuted-action-of-type type-of-action-of
    If left-clicks-available Then
        random-action-of-type click-action
    Else
        random-action-of-type-other-than drag-action
    EndIf
```

Algorithm A.20

```
If drag-actions-available Then
    random-unexecuted-action-of-type hit-key-action
Else
    random-action-of-type click-action
EndIf
```

Algorithm A.21

```
If drag-actions-available Then
    random-unexecuted-action-of-type hit-key-action
Else
    random-action
EndIf
```

Algorithm A.22

```
If drag-actions-available Then
    random-unexecuted-action-of-type hit-key-action
Else
    random-action-of-type-other-than type-action
EndIf
```

Algorithm A.23

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

Algorithm A.24

```
If
    number-of-actions-of-type click-action
    GreaterThan
    number-of-unexecuted-left-clicks
Then
    random-least-executed-action
Else
    random-most-executed-action
EndIf
```

Algorithm A.25

```
random-action-of-type-other-than type-of-action-of
    If Not
        number-of-unexecuted-drag-actions
        GreaterThan
        number-of-type-actions
    Then
        random-unexecuted-action-of-type type-of-action-of previous-action
    Else
        If random-number Equals number-of-drag-actions Then
            random-action-of-type hit-key-action
        Else
            random-action-of-type-other-than type-action
        EndIf
    EndIf
```

Algorithm A.26

```
If number-of-actions Equals number-of-type-actions Then
    random-unexecuted-action-of-type drag-action
Else
    random-unexecuted-action-of-type click-action
EndIf
```

Algorithm A.27

```
If number-of-actions Equals number-of-previous-actions Then
    random-action-of-type type-of-action-of
        If
                number-of-unexecuted-drag-actions
                GreaterThan
                number-of-actions
            Or
                number-of-type-actions
                Equals
                random-number
        Then
            random-action
        Else
            random-action-of-type-other-than
                type-of-action-of random-most-executed-action
        EndIf
Else
    random-unexecuted-action-of-type click-action
EndIf
```

Algorithm A.28

```
If number-of-actions Equals number-of-previous-actions Then
    random-action
Else
    random-unexecuted-action-of-type click-action
EndIf
```

Algorithm A.29

```
If number-of-left-clicks Equals number-of-previous-actions Then
    random-action
Else
    random-unexecuted-action-of-type click-action
EndIf
```

A.4 Other

Algorithm A.30 Original strategy in TESTAR

```
random-action
```

Algorithm A.31 The EARV strategy [17]

```
If number-of-left-clicks Greater Than number-of-type-actions Then
    random-action-of-type
        click-action
Else
    random-unexecuted-action
EndIf
```

Practical information

Student

Ir. Marion de Groot
Eerste Atjehstraat 118-B
1094 KS Amsterdam
mlmdegroot@gmail.com
+31 6 4629 0343

First mentor

Dr. Ir. Jeroen Keiren
Assistant Professor Computer Science
Faculty of Management, Science & Technology
Open University of the Netherlands
PO Box 2960, 6401 DL Heerlen
Jeroen.Keiren@ou.nl

Committee chair and second mentor

Prof. Dr. Tanja E.J. Vos
Professor Software Engineering
Faculty of Management, Science & Technology
Open University of the Netherlands
PO Box 2960, 6401 DL Heerlen
Tanja.Vos@ou.nl