The Applicability of TESTAR on Accessibility Evaluation and the Benefits of Storing Test Results in a Graph Database

Thesis for bachelor Computer Science at the Open Universiteit of the Netherlands

Floren de Gier floren.degier@xs4all.nl
Davy Kager mail@davykager.nl

May 6, 2018

Contents

List of Figures

List of Tables

PREFACE

This thesis completes our research project for the bachelor Computer Science at the Open Universiteit of the Netherlands. The project is a joined effort between Davy Kager and Floren de Gier.

The goal of the bachelor project, as proposed by our client, was to improve the presentation and visualization of the information about the tests that the automated test tool TESTAR has executed. This was prompted by the fact that TESTAR stored most of its output in large log files at the time, causing difficulties for testers to determine what errors TESTAR had found and what the test coverage was. The project proposal offered an extension with graphs as a possible solution to these problems. Our first sub-project, the research of Floren de Gier, elaborates on this solution by investigating the integration of a graph database into TESTAR. Our second sub-project, the research of Davy Kager, broadens the scope of the project proposal by investigating how TESTAR can be applied as a tool to evaluate accessibility. After clearing both ideas with our client, we successfully completed both sub-projects. That means we extended TESTAR to improve its output and widen its field of application.

The source code of our extensions to TESTAR is available online as part of the TESTAR distribution. When we started the bachelor project, the source code of TESTAR was stored in a Subversion repository hosted by the Universitat Politècnica de València. We chose to copy this repository to a private Git repository hosted by the GitHub service, because we expected the workflow with Git to be more effective than the Subversion workflow. Looking back at how smoothly the development process ran this was a sound decision. In particular, the pull request feature on GitHub allowed us to ensure that every change in the source code made by one developer was reviewed by the other developer before it was integrated. Inspired by our results, TESTAR has now officially moved to its own development repository on GitHub[1], where our code has also been integrated.

Three stakeholders from the universiteit played key roles during the course of our bachelor project. Tanja Vos is the TESTAR project lead and the client for whom we further developed TESTAR. Stijn de Gouw is our mentor. Both client and mentor iteratively reviewed our thesis throughout the creation process. Harrie Passier performs the final, independent assessment of the project results. We would like to thank these people for their efforts.

[1] https://github.com/TESTARtool/TESTAR_dev/

1

INTRODUCTION

Users of an application experience the application through its user interface. When the application is not giving the expected response or, worse, when it crashes, this will result in people not using the product anymore. So proper testing of the system *through* the user interface is very important.

Manually testing the system *through* the user interface is very labor intensive and boring, so software development teams will try to automate these tests. Various techniques exist to do that. The common techniques are writing scripts to interact with the user interface (for instance, Selenium[1]), Capture Replay (CR) [20] and Visual GUI Test (VGT) [3]. Another, less common technique is to test a system by automatically traversing the user interface, discovering the user interface components along the way. In each step the state of the user interface can be evaluated. One tool that implements this technique is TESTAR[2].

TESTAR is an automated tool that performs tests on websites, desktop software and more recently also mobile apps. The tool is developed by the Universitat Politècnica de València, Utrecht University and the Open Universiteit of the Netherlands in the context of several European projects [36]. TESTAR is also the tool that our research project revolves around.

TESTAR generates a lot of output that is poorly structured. We think that, by storing the output in a graph database, we can improve the structure and allow users to define custom queries that help them to evaluate a system.

An increasingly important property of a user interface is its accessibility. While TESTAR tests a system, not user interfaces, we believe that we can apply TESTAR to evaluate user interfaces as well. Accessibility is the property we chose to evaluate.

In this thesis, we present the results of our research where we investigated the benefits of storing test results in a graph database and realized this with an extension of TE-

[1] http://www.seleniumhq.org
[2] https://testar.org/

STAR. Furthermore, we investigated how TESTAR can be used to evaluate the accessibility of desktop software and also realized this as an extension of TESTAR. Floren de Gier investigated the benefits of storing information in a graph database and Davy Kager focused on evaluating accessibility. Both extensions will be validated using VLC[3], a real-world application.

1.1. GOALS

At the start of our research we defined our goals, each focusing on a different sub-project. In this section, we present these goals.

1.1.1. SUB-PROJECT: STORING TEST RESULTS IN A GRAPH DATABASE

The goal of this research is to improve the flexibility of TESTAR to allow the user to retrieve more information from the available data by storing the data in a structure provided by a graph database. To this end, we investigate:

Q1 How can we store the results of TESTAR in a database and is a graph database the most desirable type? (Section 3.3)

Q2 How can we retrieve extra information from an application just by storing this extra information in the customized code of the protocol? (Chapter 5)

Q3 How does recording test results for a run with TESTAR in a database and the current way of storing test results compare? (Section 9.2.1)

Q4 What is the performance impact of using a graph database during a TESTAR run? (Section 9.3)

Q5 How can we execute uniform queries on the data gathered by TESTAR? (Section 3.4.2)

1.1.2. SUB-PROJECT: ACCESSIBILITY EVALUATION

Accessibility is becoming an increasingly important software property. European standard EN 301 549 [11] contains accessibility requirements for public procurement of web content, documents, software and more. Such requirements will continue to gain importance as the United Nations Convention on the Rights of Persons with Disabilities (CRPD)[4] is ratified by more and more countries. One way to evaluate accessibility is through automated tools.

Many automated tools exist to evaluate the accessibility of web content[5]. We found relatively few tools to perform the same task on desktop software, so we put forward TE-

[3] https://www.videolan.org
[4] https://www.un.org/development/desa/disabilities/convention-on-the-rights-of-persons-with-disabilities.html
[5] https://www.w3.org/WAI/ER/tools/

STAR as a tool that can support accessibility evaluation of desktop software. To this end, we investigate:

Q6 What is a suitable accessibility standard to evaluate the accessibility of desktop software? (Chapter 6)

Q7 Which oracles can be defined for the selected accessibility standard? (Chapter 7)

Q8 How can TESTAR be extended to support accessibility evaluation? (Chapter 8)

Q9 How do experts benefit from our implementation of accessibility evaluation in TESTAR? (Section 9.4)

1.2. OUTLINE

This introduction is followed by Chapter 2 that describes the context of our research. We then enter the graph database part of the thesis.

Chapter 3 explains how test results are recorded in user interface test tools and in TESTAR in particular. This is followed by selecting the proper solution for storing the information. Chapter 4 presents the integration of the graph database within the existing architecture of TESTAR. We complete the discussion about the design of the database interaction with a discussion about extension possibilities in Chapter 5.

The next part is about accessibility evaluation, where we will show that the graph database from the first sub-project is immediately useful to support accessibility evaluation from this second sub-project. Chapter 6 introduces the concept of accessibility standards and explains our selection of a particular standard. Chapter 7 presents the automated accessibility rules that we derived from the accessibility standard. The extension of TESTAR to support accessibility evaluation described in Chapter 8 concludes this part of the thesis.

Chapter 9 brings the two sub-projects together in a case study where we apply the new features of TESTAR on the free video player VLC. Finally, we present our conclusions in Chapter 10.

2

RESEARCH CONTEXT

Our research is part of the ongoing research activities around the TESTAR[1] tool for automated testing. TESTAR autonomously explores and interacts with software through its GUI. This approach makes it possible to run tests without having to script them in advance [34]. Moreover, the GUI can change without the need to change the tests. TESTAR uses the accessibility Application Programming Interface (API) of the operating system to interact with the GUI [6]. A problem with this approach is that accessibility APIs do not always expose all user interface components [4], leading to incomplete test coverage. However, as the accessibility API is obviously tied to accessibility in addition to automated testing, this should be seen as a problem with the software being tested.

TESTAR was evaluated in a number of industry environments [7, 42, 4, 24]. The overall conclusion is that automated testing at the GUI level is a promising field of research. But some concerns also emerged, such as the need for better documentation and the difficulties inherent to the trial-and-error process of defining oracles. The latter probem in a general form, in this case in the context of software with a GUI, is the oracle problem. Oracles must distinguish between correct system behaviour and behaviour that is incorrect or unexpected for a given test, but defining appropriate oracles is non-trivial [44]. Indeed, we devote an entire chapter (Chapter 7) on the subject of defining accessibility oracles.

TESTAR is a test tool that can be categorized as a crawler or GUI ripper [1]. By dynamically traversing a GUI, a model of the software is created. Other types of test tools are VGT and CR tools. A big advantage of TESTAR compared to these other tools is the fact that TESTAR does not require test scripts to be created and maintained. Especially the maintenance is a big hurdle when development teams introduce automatic testing [19, 2]. There are similar tools that follow the same approach as TESTAR, like Guitar[2] and Murphy[3]. However, these tools do not use the accessibility API of the operating system. Using the accessibility API makes TESTAR independent of the technology used to implement the software being

[1] https://testar.org/
[2] https://sourceforge.net/projects/guitar/
[3] https://github.com/F-Secure/murphy

tested.

2.1. EXECUTION FLOW

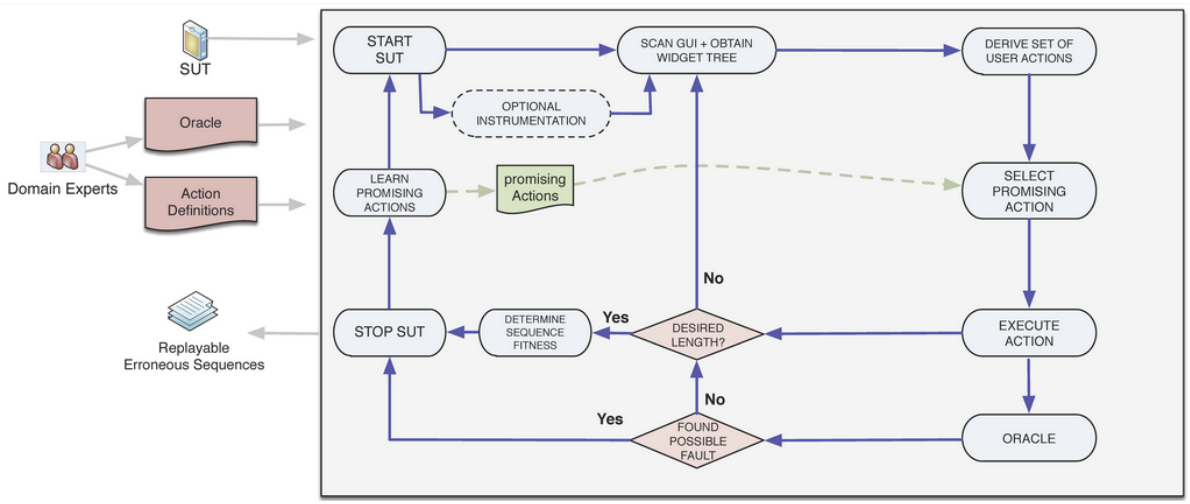


Figure 2.1: TESTAR execution flow

Figure 2.1 shows the execution flow when running TESTAR [42]. There are two nested loops. The outer loop corresponds to the complete test suite. This loop runs test sequences. The inner loop corresponds to a single test sequence. This loop runs tests and derives actions.

TESTAR first starts the software, referred to as SUT. Then, TESTAR gathers the current state of all user interface components, referred to as widgets, storing the information in a tree data structure. From this tree, the state of the SUT can be examined for problems and the possible actions to execute can be derived. One action to execute is selected using an action selection algorithm, either randomly or through a more sophisticated method such as learning metaheuristics [13] or genetic algorithms [14]. After executing the action, TESTAR goes back to crawling the GUI to build a refreshed tree.

TESTAR continuously executes actions until the stop criteria for the test sequence are met. At that point, TESTAR stores the test results and shuts down the SUT. This is where a new sequence begins, if any. Otherwise, the test suite finishes. The stop criteria are customizable.

TESTAR uses test protocols to execute the tests. The point where TESTAR is unique compared to other test tools is the fact that the protocols are extensible by the user. TESTAR comes with a protocol editor. It also provides a series of pre-made protocols for common scenarios. Figure 2.2 shows the structure of the protocols.

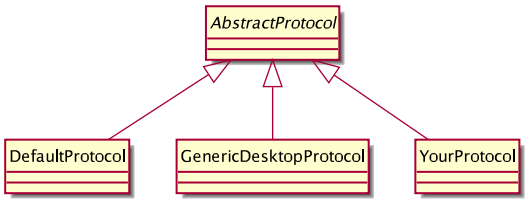


Figure 2.2: Class diagram showing the relation between the specific protocols and the `AbstractProtocol`

2.2. STORING TEST RESULTS

Our first sub-project will focus on the test results recorded by TESTAR. The minimum amount of evidence a test can generate is that it runs without crashing. Although this might be useful, it is good to have some information about the tests that were executed. Most test tools cater for this. As an example, Listing 2.1 shows the results stored by the Java tool JUnit[4].

Listing 2.1: JUnit test results

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="com.altran.igjava.cucumber.DemoTest" tests="2" skipped="0"
      failures="1" errors="0" timestamp="2017-12-03T11:39:07"
      hostname="MarvinMobile.fritz.box" time="0.01">
 <properties/>
 <testcase name="validateHello2World" classname="com.altran.igjava.cucumber.DemoTest" time="0.006"/>
 <testcase name="validateHelloWorld" classname="com.altran.igjava.cucumber.DemoTest" time="0.003">
  <failure message="org.junit.ComparisonFailure: expected:&lt;Hel[]o World&gt; but was:&lt;Hel[l]o World&gt;" type="org.junit.ComparisonFailure">
   org.junit.ComparisonFailure: expected:&lt;Hel[]o World&gt; but was:&lt;Hel[l]o World&gt;
   at org.junit.Assert.assertEquals(Assert.java:115)
   at org.junit.Assert.assertEquals(Assert.java:144)
   ...
  </failure>
 </testcase>
 <system-out><![CDATA[Test OK]]></system-out>
 <system-err><![CDATA[]]></system-err>
</testsuite>
```

Listing 2.1 shows a test (the `testsuite`) that consists of two `testcases` ("validateHello2World" and "validateHelloWorld"). For the `testsuite`, it shows some metrics like the number of tests in the suite, the date of execution, the number of tests skipped or failed and the number of tests that ended up in an error. For each `testcase` the execution time is recorded and for the failed tests a failure message is recorded.

Although the example presented in Listing 2.1 shows the results of a low-level tool that performs a different type of test compared to the high-level test tools that interact with the GUI, we can pick the key elements of the output that every test result should contain. These key elements are described in Table 2.1.

Element	Description
What	The name of the test suite and test cases should be descriptive enough to give the user an idea of what the test is about.
When	The timestamp is important to relate the results to a specific version of the SUT.
Results	The evidence if the test is passed or failed and additional information recorded during the test execution.

Table 2.1: Key properties of test results

Chapter 3 will show how TESTAR and two other GUI test tools present the elements presented in Table 2.1 in their output.

[4] http://junit.org

2.3. ORACLES AND VERDICTS

The test protocols in TESTAR take care of starting the SUT, running the test sequences, collecting information about the tests and finally shutting down the SUT. Most interesting are how the tests are run and what information is gathered. Section 2.2 described the evidence of a test as a key property. Important concepts in the test protocol are oracles and verdicts. An oracle tests if the result of a test matches an expected value. A verdict is the evidence or outcome of an oracle.

An oracle consists of two parts [44]:

Oracle information Describes the expected test result.

Oracle procedure Compares actual test output to the oracle information.

A verdicts is the result of an oracle procedure. A verdict can represent either a match or a mismatch between the actual test output and the oracle information. The first kind means that the actual behaviour matched the expected behaviour. The second kind is a problem. We define a problem in the context of TESTAR as a mismatch between actual and expected behaviour of a SUT as decided by the oracle procedure based on the oracle information. In contrast, a pass is defined as a match, that is, a non-problem.

In TESTAR, verdicts have a severity between 0 and 1, where 0 indicates a match and everything higher is a problem. There is a configurable threshold above which the problem optionally leads to the test sequence being halted. Such problems are called faults in TESTAR. Verdicts below this threshold will not halt the test sequence and can be regarded as warnings or less severe problems.

2.4. ACCESSIBILITY VERDICTS

Our second sub-project will focus on accessibility evaluation. Accessibility refers to the ability to access places, products, services and information by users with and without disabilities. Accessibility covers many areas. It can apply to geographical environments like cities, or to buildings, but also to books, media, websites and software.

A particular type of testing that we investigate is accessibility testing, generally referred to as accessibility evaluation. Specifically, we want to evaluate software that the user has to operate through a GUI and that runs natively on a computer with the Windows operating system, for example a laptop or a tablet, but not a smartphone with a “Mobile” or “Phone” version of Windows. Since the evaluation can later be extended to other operating systemsn, we choose to refer to this kind of software as desktop software.

In order to distinguish the general problems defined in Section 2.3 from accessibility problems, we define the latter as violations. A further distinction can be made between definite violations and violations that require verification by an expert. This distinction is useful to classify oracles and to structure the evaluation results.

More precisely, we define errors and warnings in the context of accessibility evaluation with TESTAR as follows:

Error A definite violation that can be signalled fully automatically by TESTAR.

Warning A potential violation that can be signalled only semi-automatically by TESTAR because it requires verification by an expert.

With the descriptions, key properties and definitions from this chapter we are now ready to present our individual research sub-projects.

3

STORING TEST RESULTS

In this chapter, we investigate the output produced by TESTAR. First, we look at the current output of TESTAR and compare it with the output of two commercially available tools (see Section 3.1). Next, we present the the limitations of TESTAR and introduce a number of improvements (see Section 3.2). One of these improvements, storing results in a database, will be elaborated on. We complete this chapter by selecting the tools used in the case study which is presented in Chapter 9 (see Section 3.3 and Section 3.4).

3.1. CURRENT SITUATION

To compare TESTAR to other tools we will look at two different tools that test an application at a GUI level. We select a VGT tool (EyeAutomate[1]) and a CR tool (Rational Functional Tester (RFT)[2]). The results are summarized in Table 3.1. First, we zoom in on the test output of TESTAR.

3.1.1. TESTAR

TESTAR generates a lot of information during its execution. This information is stored in separate folders for different types of data. These folders are:

log (text) files

graphs which can be transformed into graphical presentations using dot [3]

sequences (an internal binary format) (split in OK, suspicious, unexpected close)

metrics information (Comma-Separated Values (CSV)).

[1] http://eyeautomate.com
[2] https://developer.ibm.com/testing/functional/
[3] https://graphviz.gitlab.io

Looking at the set of essential elements for proper test output that we introduced in Section 2.2, TESTAR does not completely comply. Meta-information, specifying *what* information is stored and *when*, is found in a text file which is recorded in the output folder. The name of this file decodes *when* the test is executed and the contents show basically *what* is being tested. The *results* are available in the different sub-folders of the output folder as mentioned at the beginning of this section. The data in these folders is ordered by the sequence number of the sequences executed by TESTAR. TESTAR only looks at the highest sequence number to determine the next sequence number despite the fact that the test executed in the next sequence can have a completely different definition.

3.1.2. OTHER TOOLS

In order to put things in perspective, we looked at EyeAutomate and RFT to see how these tools store their test results. In this section, we provide a brief summary.

EyeAutomate [15] is a VGT tool. The tool allows the user to define a test script as a combination of template steps enclosed within a "Begin" and "End" statement to indicate the test steps. These templates are things like opening websites, clicking on buttons, etcetera. The output consists of a series of HyperText Markup Language (HTML) files stored in the "reports" folder. The name of these files contains the name of the test with the time of execution appended. If these files are viewed in a web browser, it is possible to view the screenshots for each test step. Figure 3.1 shows an example of a test report from EyeAutomate.

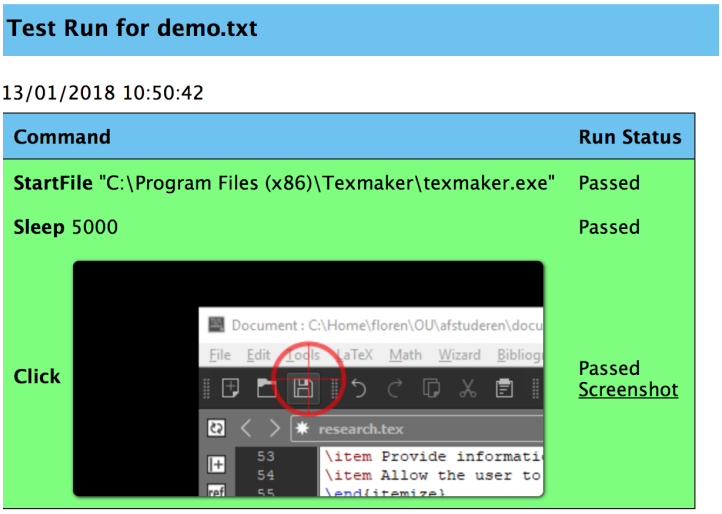


Figure 3.1: Example of a test report from EyeAutomate

RFT is a CR tool we investigated. RFT allows the user to test a Java application by replaying recorded sequences. These recordings result in test scripts which can be edited by the tester. At their website [20] they describe that the test results are ordered within the project tree. A log viewer is available to view the results. It is also possible to view the results in a web browser.

Both tools facilitate a replay mode just by executing a test again. Looking at the set of essential elements for proper test output that we introduced in Section 2.2, we can conclude that both tools reflect *what* is being tested (the name of the test script is part of the result file), *when* the test has been executed (showing a timestamp in the name of the result file) and what the *results* are (for each step the output shows if the step passed or failed). However, besides showing the fact that a test is passed or failed, the results do not provide a lot of possibilities for further analysis apart from test step execution time.

Table 3.1 provides an overview which shows how the three tools discussed in this chapter implement the key elements introduced in Section 2.2.

	RFT	EyeAutomate	TESTAR
what	The test cases are described in "test script". These scripts are small Java programs that perform actions on the GUI of the SUT.	The test cases are described in a simple text file which can be created in the tool or in a text editor.	Test cases for TESTAR are described by the settings file and the protocol class. In the protocol class, pass or fail criteria can be defined as well as the way TESTAR "walks" through the SUT.
when	RFT puts the test results in a folder which has the timestamp of the test execution as date.	EyeAutomate provides an overview of the executed tests including the times of execution.	For each test execution, TESTAR creates a file "<timestamp>.log" in the root of the output folder. This file shows when the test is executed and the settings used during the execution.
results	The results of RFT are provided as a HTML file, providing an overview, and a set of CSV files with detailed information	EyeAutomate presents the test results as a HTML file showing each test step with a screenshot and a statement if the step is passed or failed.	TESTAR provides various types of test output ranging from text files to binary artefacts containing the data required to replay a certain sequence. The files are not labelled with the time of execution. Instead, they are labelled with the number of the sequence they belong to.

Table 3.1: Overview of the comparison of different GUI test tools

3.1.3. COMMON PROPERTIES

Having looked at three test tools focusing on their output, we can conclude that the output of the different tools have some common properties:

- Provide an overview of the test results in a web-based presentation.
- Allow the user to view screenshots from failed scenarios.
- Provide information about what is tested.
- Provide information about when the test is executed.
- Allow the user to replay tests.

TESTAR more or less provides all these features except for a web-based overview of the test results. In the next section we zoom in on other limitations of TESTAR.

3.2. LIMITATIONS OF TESTAR

Relating the different pieces of information, listed in 3.1.1, is possible since the basic pieces of information, states and actions, have a unique name. However, since the information is currently stored in different files and in different formats, it is quite cumbersome to achieve a unified approach. This problem is further compounded by the large number of steps and sequences required to analyse a complete application. Another problem is that it is hard to gather custom meta-information (like CPU usage) without having to change the TESTAR source code. Finally, the data recorded by TESTAR does not have a uniform format, which makes it hard to write a query using all recorded data.

3.2.1. SUGGESTIONS FOR IMPROVEMENT

Having looked at the output of TESTAR and its limitations, we can suggest a couple of improvements:

S-1 Provide an overview of the test results in a format that can be viewed in a web browser.

S-2 Move the test results for a single test execution into a separate folder.

S-3 Store the model discovered during the test execution in a database.

In the remainder of this repoty we will elaborate on S-3 and extend TESTAR with the capability of storing the model data in a database besides the existing file-based solution. In the next section we will select the proper database solution.

3.3. TOOL SELECTION

One of the main goals of this research is to investigate the benefits of storing the results of TESTAR in a database (Question Q1), implementing Suggestion S-3. Putting the information in a database can be beneficial in the following cases:

B-1 It can help to identify the root cause of a failing test since the relation between the actions and states is available in the database.

B-2 It opens up the possibility to perform post-test analyses on the test results by generating custom queries. For instance the number of widgets found during the test and the amount of widgets on which an action was applied. This can help users of TESTAR to evaluate the selected action selection algorithm. The current solution provides some metrics, but they cannot be altered without changing the application code.

B-3 Since all states are stored in the database, it is possible to create a query which uses the information from a sequence of states. TESTAR oracles only act on a single state.

B-4 Queries on a database provide a more user-friendly way to gather information about the state of the SUT. Oracles in TESTAR need to be implemented in low-level Java code.

This section will present the selection of the database technology used within the context of our research as well as the technology used to retrieve information from the database.

3.3.1. DATABASE SELECTION

In order to optimally exploit the benefits of storing test results in a database, the appropriate database technology has to be selected. The technology should be sufficiently flexible to satisfy the following requirements:

REQ-1 The database should reflect the relation between different states. (providing the Benefits B-1, B-2 and B-3).

REQ-2 The database should reflect the relation between a state and the active widgets in that state. (providing Benefit B-2).

REQ-3 The database should allow the user to retrieve the execution path of the application. (providing Benefit B-1).

REQ-4 The database should store a variable number of properties for a given state, widget or action. (providing Benefits B-1 through B-4).

REQ-5 The database technology shall provide means to query the data. (providing Benefit B-4).

REQ-6 The selected technology shall be open source since TESTAR is also open source.

The website of DB-Engines [35] provides a series of definitions of possible database types. Considering our requirements and the presented definitions, the database should implement a Multi-model approach reflecting both the properties of a Graph Database Management System (DBMS) and a Key-Value store. Looking at the ranking at DB-Engines, we see the following top 3 of graph database solutions:

1. Neo4j
2. Microsoft Cosmos DB
3. OrientDB

Microsoft Cosmos DB (see `https://cosmosdb.com`) is a cloud solution which is not free to use and always requires internet access. Currently, TESTAR does not require the system to be connected to the internet. For this reason, Microsoft Cosmos DB is not further investigated.

Neo4j is the number one graph database in the ranking. Neo4j is not a key-value store and this makes it harder to comply to Requirement REQ-4. This is the main reason OrientDB is selected over Neo4j.

Being a graph database, OrientDB meets REQ-1, it is possible to present the relation between the different states through the path; "State has a Widget on which an Action can be executed" that results in a new State. REQ-2 can be met by modelling the "has" relation between the state vertex and the widget vertex as an edge. REQ-3 can be met using a traversal engine (see Section 3.4.2) to analyse the execution path. Since OrientDB is also a key-value store, it is possible to store arbitrary properties with a vertex or an edge. This capability demonstrates that OrientDB also meets REQ-4. OrientDB is available as a free, open source product and thus compliant to Requirement REQ-6. OrientDB provides multiple tools to query the database and therefore meets Requirement REQ-5. OrientDB is selected.

3.4. GRAPH DATABASES

Having selected a graph database as our storage solution (see Section 3.3.1), we will now briefly discuss the properties of a graph database (see Section 3.4.1). This discussion is followed by an explanation of traversal engines which can be used to analyse the data in a graph database (see Section 3.4.2).

3.4.1. REPRESENTING MODELS IN A GRAPH DATABASE

Angles et al [5] present two different types of graph models: the edge-labelled graph and the property graph. Within the edge-labelled graph all properties are modelled as edges. If this model would be applied to TESTAR, every property will be modelled as a separate vertex, containing the value as its id, connected to the State through an edge which contains the type of the property as its label. Gathering all properties of a State requires the user to

traverse through all these edges. A property graph allows us to model the properties of a State, Widget or Action, as key-value pairs on a vertex or edge. OrientDB follows the model of the property graph. Consequently, The property graph model is a better match for our purpose.

3.4.2. TRAVERSAL ENGINE

Retrieving information from a graph database like OrientDB can be done, among others, using a graph traversal language. A graph traversal language is, as stated by name, a language which can be used to traverse through a graph to gather information from the data. There exist various languages, such as SPARQL[4], Cypher[5] or Gremlin[6].

Since OrientDB is selected as the graph database, we are limited to the traversal languages supported by OrientDB. OrientDB provides the following possibilities to traverse the graph:

1. Traverse the graph using the Structured Query Language (SQL) dialect of OrientDB.
2. Traverse the graph using the Traverse capability of the Java API.
3. Traverse the graph using Gremlin.

Both option 1 and 2 are only applicable to OrientDB. In our research, we are looking for a solution which is not vendor-specific (Question Q5). Therefore, Gremlin will be used as the traversal language to query the database. In [33] the mathematical properties of Gremlin are described. The formalization from that paper will be used in the further discussion in this section.

BASIC OPERATION OF GREMLIN

To understand the way Gremlin works, we first define the key elements of a Traversal. As explained in Chapter 2 of [33], three elements make up the graph traversal machine. We list these three elements with their reference in [33]:

- The **Graph** G defined as $G = (V, E, \lambda)$ where V is a set of vertices, E the edges between the vertices and λ represents the key-value pairs that make up the properties of the vertices and edges. (see Section 2.1)
- The **Traversal** ψ defines the query which is applied to the Graph G. (see Section 2.2)
- The **Traverser** T represents the execution of the query. (see Section 2.3)

[4] https://www.w3.org/TR/rdf-sparql-query/
[5] http://www.opencypher.org
[6] https://tinkerpop.apache.org/gremlin.html

A Traversal consists of a number of functions, called steps, which form a linear chain ($f \circ g \circ h$) where the output of f is the input for g and the output of g is the input for h. The functions of a traversal can also be nested. For instance in the Traversal $f(g \circ h) \circ k$), where f uses the result of $g \circ h$ as its input and the output from f is used as input for k. Rodriguez [33] identifies 5 types of functions (map, flat map, filter, side effect and branch). All these types can be represented as a flat map. Rodriguez gives the following definition of a flat map function:

> "$A^* \rightarrow B^*$, where the output Traverser set may be smaller, equal to, or larger than the input Traverser set."

From this quote it is clear that a function translates one Traverser set to another set of traversers. Using the Graph and a number of steps, we can define a Traversal on the recorded data:

$$G.V \circ has(\text{“}@class''\text{,“}Widget'') \circ map(\text{“}Desc'') \tag{3.1}$$

Equation 3.1 shows a Traversal that consists of a couple of steps which are arranged in a linear motif. The first step V takes the Graph and creates a Traverser set A^* consisting of a Traverser for each Vertex. The second step ($has(\text{“}@class''\text{,“}Widget'')$) reduces A^* to a set B^* of Traversers for each Vertex of type "Widget". The last step, ($map(\text{“}Desc'')$), creates a traverser set C^* consisting of the same number of elements as B^*. However, the objects now only contain the "Desc" property. In the case study presented in Chapter 9, we will use Gremlin to retrieve information from the graph database.

LIMITATIONS FOR THE USAGE OF GREMLIN WITH ORIENTDB

During our research, we used OrientDB version 2.2.29. This is the stable version at the moment of writing. The Gremlin version supported is an older version which is not as complete as the current version published at `http://tinkerpop.apache.org`. OrientDB provides its own Gremlin console which implements an older version of Gremlin described at `https://github.com/tinkerpop/gremlin/wiki`. This later version is used throughout this research. Starting at OrientDB version 3.x, OrientDB will provide a plug-in for Gremlin. This allows the user to query OrientDB through the standard Gremlin version.

3.5. CONCLUSION

In this chapter we investigated the output of TESTAR and compared this output with the output of a selection of commercially available tools that test the application at a GUI level. We can conclude that TESTAR provides all the data also provided by the commercial tools. Since TESTAR is a tool which does not rely on scenarios or scripts to define the test sequences, it is important that the tool provides replay capabilities. This feature is part of TESTAR. TESTAR does not label the result data in a way that it can be related to a test run simply by looking at the file names. Hence we suggested in Suggestion S-2 to put the results in separate folders for each test run. TESTAR is lacking a nice web-based report providing a summary of the results which can be navigated to provide more detail as suggested in Suggestion S-1.

Although TESTAR stores a lot of information, this information is not stored in a uniform way. Furthermore, the information is stored in different files and cannot be used to create oracles. Storing information in a graph database can solve these shortcomings.

In this chapter we elaborated on Suggestion S-3 to add support for a database that can be used to store the model. In Section 3.3.1 we defined a set of requirements and in the following section a proper database implementation was selected.

In Chapter 4 we present a design for the integration of the graph database within TESTAR. In Chapter 9 we evaluate our choices in a case study.

4

Design of the Database Integration

To export the results of TESTAR to a graph database, the current data model of TESTAR needs to be known. With the knowledge of this model, we create an "export" data model to export the results of TESTAR to a graph database in such a way that the stored information can be used for off-line analysis of the SUT.

We first discuss the current situation in Section 4.1. Next, we present a conceptual data model for the graph database in Section 4.2. Finally, we present the implementation of the `graphdb` module in Section 4.3.

4.1. Current Data Model

The current data model of a SUT in TESTAR consists of the following key types:

State contains the state of the SUT at a given step within the execution of the test. `State` can be used to discover the active widgets at a given moment.

Widget is the representation of a visible object of the application. This can be a menu item, a title bar, a dialogue, etcetera. For TESTAR, only the widgets that are currently available for manipulation (active) are important.

Action is the type that is constructed by TESTAR from a `Widget`. `Action` describes the way to interact with the `Widget` and is used by TESTAR for instance to push a button or to fill a text box.

For a more detailed description, see appendix A and B of the TESTAR user manual [37].

4.1.1. STRUCTURE

Figure 4.1 shows the key elements[1] and their relation. A `Widget` is always related to a `State` and an `Action` can target a `Widget`. When an `Action` does not target a `Widget` it is applied to a `State` (for instance when "Escape" is the `Action`). Tables 4.1, 4.2 and 4.3 describe the properties of the model.

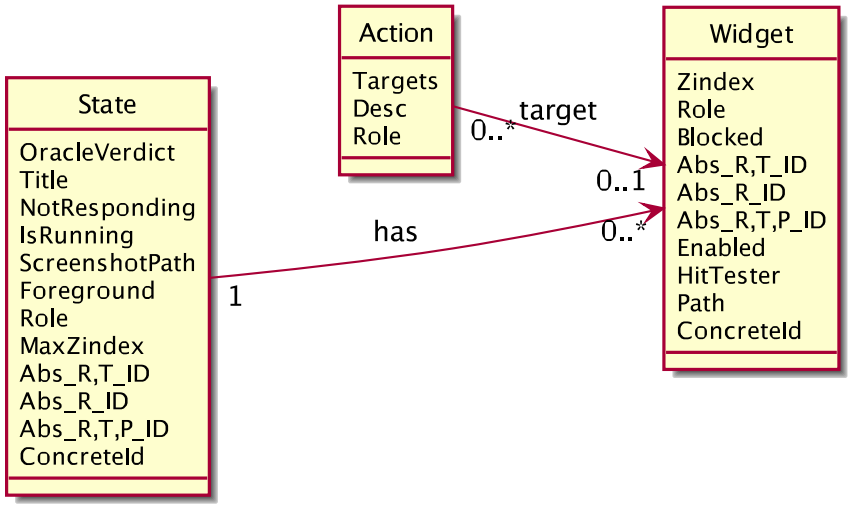


Figure 4.1: Structure of the internal model of the SUT constructed by TESTAR during test execution. Note: Only the properties that are relevant for our research are listed.

Property	Description
Targets	The ConcreteID of the target `State` of the `Action`
Desc	A textual description of the `Action`
Role	The role of the `Action`

Table 4.1: Description of the objects for the `Action` type

[1] State, Widget and Action all have more elements. For instance all properties discovered by the accessibility API are stored.

Property	Description
OracleVerdict	Textual representation of the Oracle
Title	A textual description of the `State`
NotResponding	Indicates if the SUT was responding in the given `State`
IsRunning	Indicates the SUT was running in the given `State`
ScreenshotPath	Location on the file system where the screenshot for the `State` is stored.
Foreground	Indicates the SUT was running as a foreground process.
Role	The role of the `State`
MaxZindex	The Z-index for the window that is on top. This identifies the active area of the application.
Abs_R,T_ID	The abstract ID that uniquely identifies the `State` by its role and title.
Abs_R_ID	The abstract ID that uniquely identifies the `State` by its role.
Abs_R,T,P_ID	The abstract ID that uniquely identifies the `State` by its role, title and path.
ConcreteID	Unique ID of the `State`.

Table 4.2: Description of the objects for the `State` type

The abstract IDs are the same for `Widget` or `State` objects when they share the same values for the properties used to construct the ID. For instance when two `Widgets` have the same role, they will have the same Abs_R_ID.

Property	Description
Zindex	The Z-index for the Widget. If the Z-index is equal to the MaxZindex of the State, the Widget can be selected in that State.
Role	The role of the Widget.
Blocked	Indicates if the Widget is blocked (for instance by another modal dialogue).
Abs_R,T_ID	The abstract ID that uniquely identifies the Widget by its role and title.
Abs_R_ID	The abstract ID that uniquely identifies the Widget by its role.
Abs_R,T,P_ID	The abstract ID that uniquely identifies the Widget by its role, title and path.
Enabled	Indicates if the Widget is enabled.
HitTester	Class used to check if a Widget is reachable with the mouse (not obscured by other widgets).
Path	Indicates how the Widget can be located in the Widget tree which is constructed by the operating system.
ConcreteID	The unique ID of the Widget.

Table 4.3: Description of the objects for the Widget type

4.1.2. BEHAVIOUR

In every step performed during the test execution, TESTAR determines which actions can be executed using a configurable selection algorithm. To do this , TESTAR first determines which widgets are reachable by looking at the Z-index[2]. For each available Widget, the protocol creates an Action if the Widget can be manipulated in the protocol. For instance, in the generic desktop protocol, only clickable (buttons, menu items, etcetera), typable (text fields) and draggable (sliders) widgets will result in an Action. In the default protocol, a Widget can only have one Action defined. However, Widgets can have multiple Actions, for instance a mouse-click and a keyboard shortcut. From these actions, an Action is selected using the chosen selection algorithm.

TESTAR maintains an internal model of states and actions in an instance of the class TestarEnvironment (see Figure 4.2). The information stored is used in certain action selection algorithms. Since the implementation of the graph database is independent from the mechanism that selects the actions, a further elaboration of TestarEnvironment and its role within TESTAR is left to the reader.

[2]The information displayed on a computer screen is modelled in 3D. The X and Y dimensions are visible. The Z dimension determines how various windows overlap each other. The Z-index of widgets presents the position in the stack of windows. The window with the highest Z-index is on top.

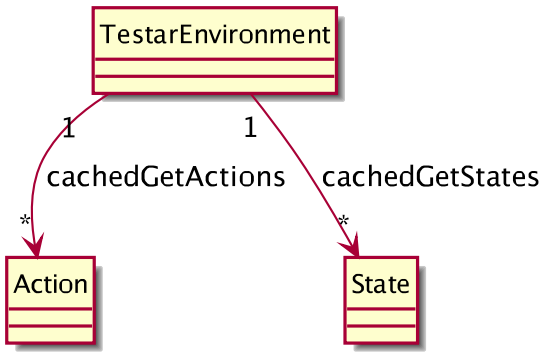


Figure 4.2: `TestarEnvironment` and its relation to `State` and `Action`

THE TAG MECHANISM

Before we continue with the discussion of the graph database model, we will explain the `Tag` mechanism used within the TESTAR code. A `Tag` is a type which can store a value in a generic way. In this section we will explain the mechanism using an example. Say we want to store the CPU temperature for each executed `Action`. First we need to define a `Tag`. Listing 4.1 shows how this is done.

Listing 4.1: Definition of a Tag

```
Tag<Integer> CpuTemperature = Tag.from("cpuTemp",Integer.class);
```

In the design presented in Section 4.3, the objects which are stored all extend the class `TaggableBase`. `TaggableBase` provides the API to set a `Tag`. Figure 4.3 shows the relation between the types stored and the `TaggableBase`. Using the `set` method from the `TaggableBase` class, we can store a value for the CpuTemperature with an `Action`. Listing 4.2 shows an example.

Listing 4.2: Sample code that sets the CPU temperature with an `Action`

```
action.set(CpuTemperature,100);
```

Now that the CpuTemperature is set, it will be stored as property to the database.

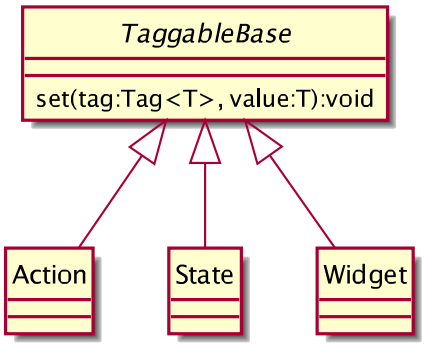


Figure 4.3: Class diagram demonstrating the relation between `Action`, `State` and `Widget` and their abstract parent `TaggableBase`

4.2. THE GRAPH DATABASE MODEL

The model as presented in Figure 4.1 will be used as the starting point for our new model in the graph database. If we want to retrieve all information which is currently available in the output of TESTAR, we have to extend the model of Figure 4.1 with a few extra properties. We will add a property to `State` indicating whether the `State` is the first (initial) state or not. We will add a property with information on the resource consumption to each executed `Action`. Finally, we add a property containing information about the representation of the action to the `Action`. This last modification allows us to print detailed information of the executed actions. In the original output of TESTAR this information is presented in the file "sequence<number>.log".

Figure 4.4 shows the resulting model. In order to keep the image clean we omit most of the edges to the abstract vertices from the diagram. The following list provides some explanation of the items presented in the figure.;

- In our graph database model, we model the `State` of the SUT as a `Vertex`.
- A `State` vertex has an outgoing `Edge` to a `Widget Vertex` for each reachable `Widget` from that `State`. Only the widgets which can be manipulated by TESTAR are present. Currently the protocol class defines which widgets are visible.
- The `State` and `Widget` vertices are connected with an `Edge` with the label "has".
- When an `Action` is executed, this `Action` is modelled as a `Vertex`.
- The `Action` vertex has an incoming `Edge` from the `State` or `Widget` it targets, labelled "targetedBy", and an outgoing `Edge` to the resulting `State` which is labelled "resultsIn".
- The graph also contains a series of "abstract" vertices. These are used to model common properties of `Widgets`, `States` or `Actions`. For instance, all `Widgets` that are list items belong to the same role. These `Widgets` are all connected to the same `AbstractRole` vertex with an edge labelled "role", `Widgets` with the same role and title are connected with an edge labelled "roleTitle" to a common `AbstractRoleTitle Vertex`, etcetera.

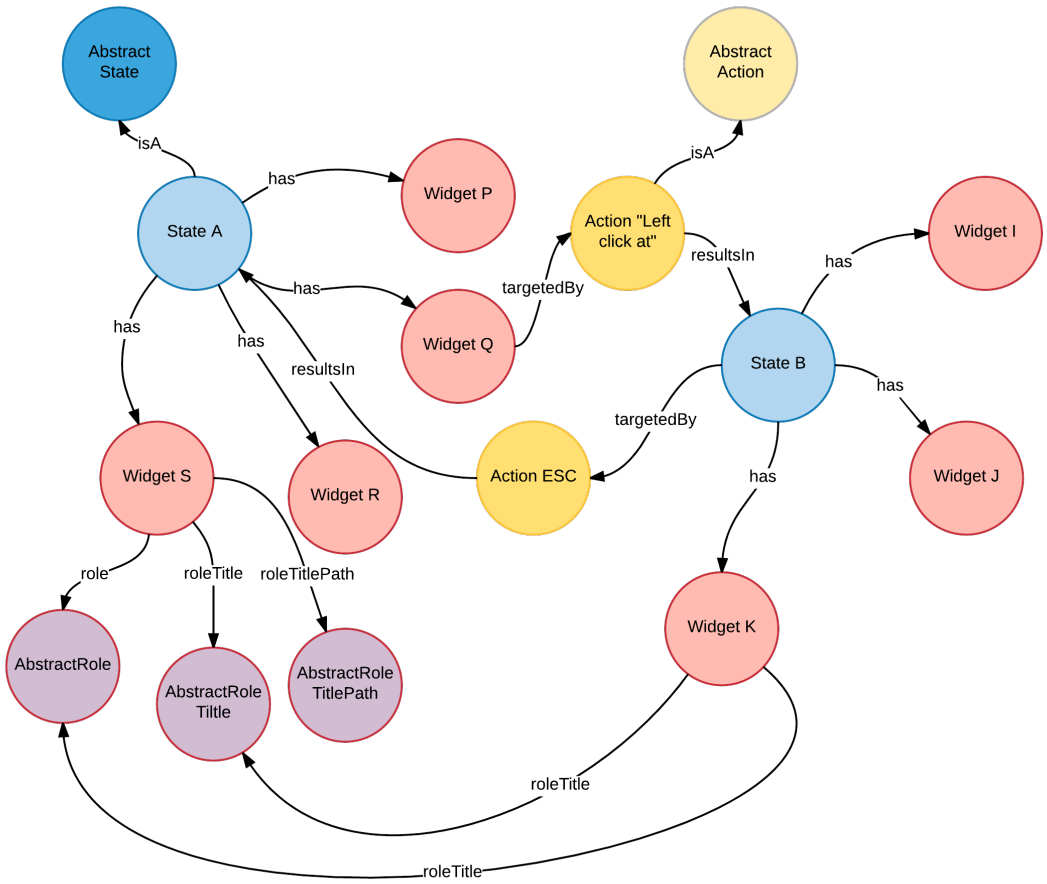


Figure 4.4: The graph model

4.3. IMPLEMENTATION OF THE MODULE GRAPHDB

The model, described in Section 4.2, is implemented as a new module (`graphdb`) within the code base of TESTAR. This section presents the structure (Section 4.3.1), how TESTAR interacts with the new module (Section 4.3.2) and finally the behaviour of the module (Section 4.3.3).

4.3.1. THE STRUCTURE OF THE MODULE GRAPHDB

Figure 4.5 shows the class diagram of the new module `graphdb`. This module is responsible for storing data from TESTAR in a graph database. The component is structured in such a way that the implementation details of the selected graph database are hidden from the logic of TESTAR. To achieve this, the TESTAR code only interacts with an instance from the class `GraphDB`. This class acts as a proxy [18]. It only passes information to an instance of the class if output to the graph database is enabled (see Appendix A).

`GraphDBRepository` is the interface implemented by both `GraphDB` and `OrientDBRepository`. The class `OrientDBRepository` provides the implementation for an OrientDB database. If

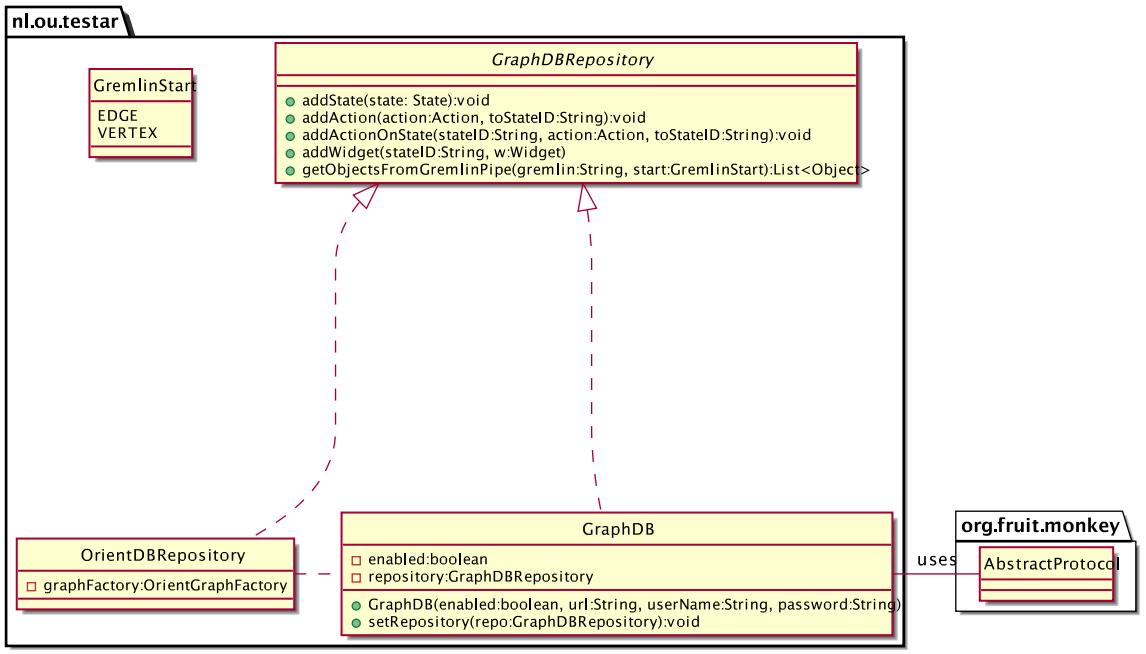


Figure 4.5: Class diagram of the `graphdb` module

support is added for other graph databases, only a specific implementation for these databases should be added as an implementation of `GraphDBRepository`. The enumeration `GremlinStart` is used together with the method `getObjectsFromGremlinPipe` to tell the method the analyses has to start at an edge or a vertex.

The interface `GraphDBRepository` contains a method, `getObjectsFromGremlinPipe`, which breaks the idea of a technology independent interface. This method was introduced at a later stage than the original design. It should be moved to a separate interface. This change is left as future work.

4.3.2. INTERACTION BETWEEN TESTAR AND THE MODULE GRAPHDB

Storing data into the graph database is implemented in the `runAction` method of the class `AbstractProtocol`. During each iteration the `State` is stored. Within the execution of `runAction` the method `deriveActions` is called which is part of the specific protocol. So the user of TESTAR should determine which widgets to store for their specific protocol implementation. We implemented this selection by storing the `Widget` for every possible action discovered in the method `deriveActions`. Storing the widgets can only be done once the protocol implementation is chosen since the available widgets are only known at the level of the specific protocol. Figure 4.6 shows the sequence diagram of a TESTAR run where only the relevant actions are included in the diagram.

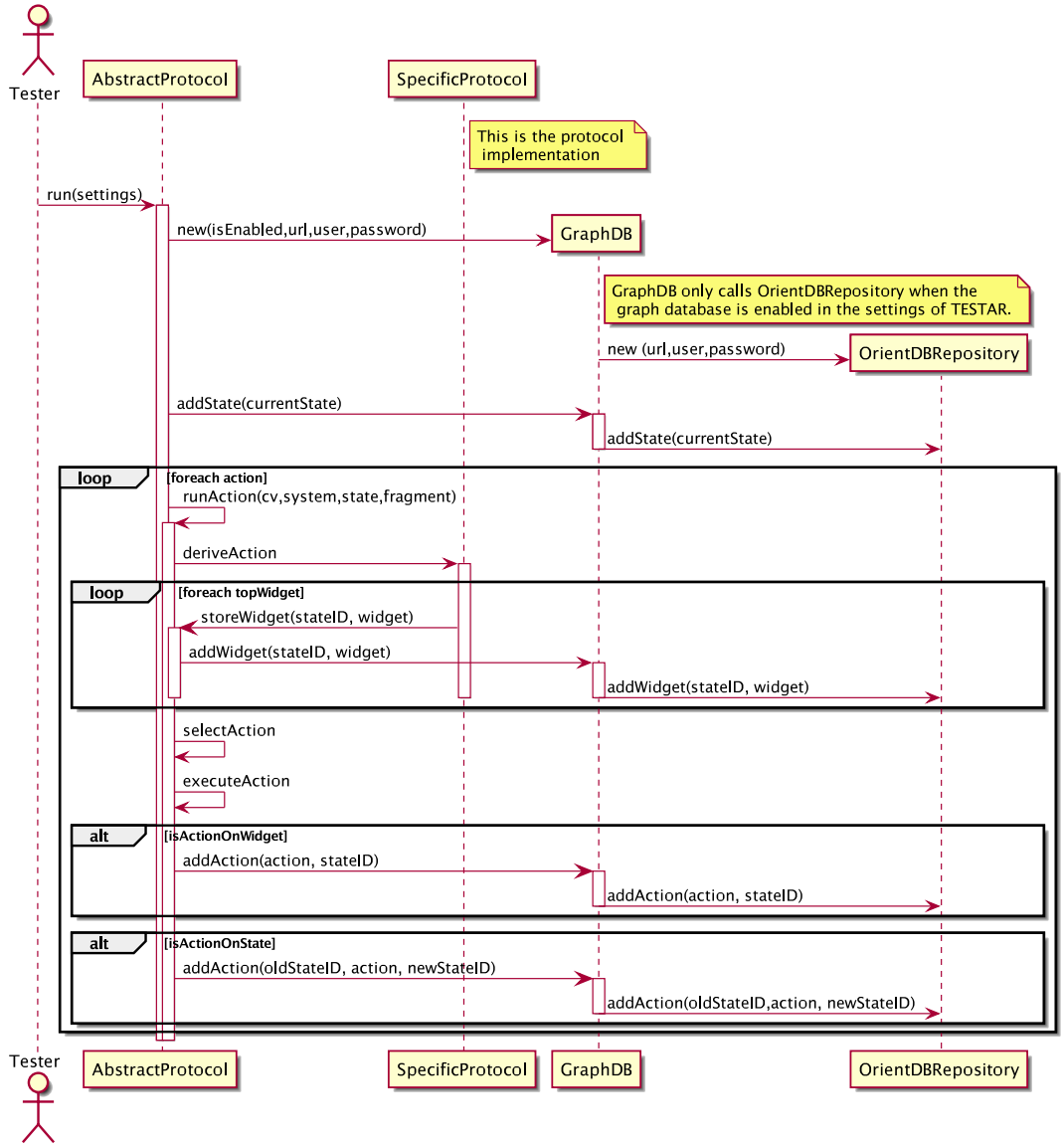


Figure 4.6: Sequence diagram demonstrating the interaction with the module `graphdb`

4.3.3. BEHAVIOUR INSIDE THE MODULE GRAPHDB

We conclude this chapter with a brief description of the behaviour For the design, presented in this chapter, it is important to explain how exactly the data of TESTAR is stored in the database. To support storage, the interface `GraphDBRepository` provides four methods. In this section, we present a sequence diagram for each of these methods.

Figure 4.7 shows the sequence for the method `addState`. Important to note is that states are only stored once. All objects, `State`, `Widget` or `Action`, are stored using their "ConcreteID" as unique key.

Figure 4.8 shows the sequence for the method `addWidget`. First the `State` object is retrieved. Next the `Widget` is retrieved. If it exists nothing happens, if it does not exist, a `Widget` vertex is created.

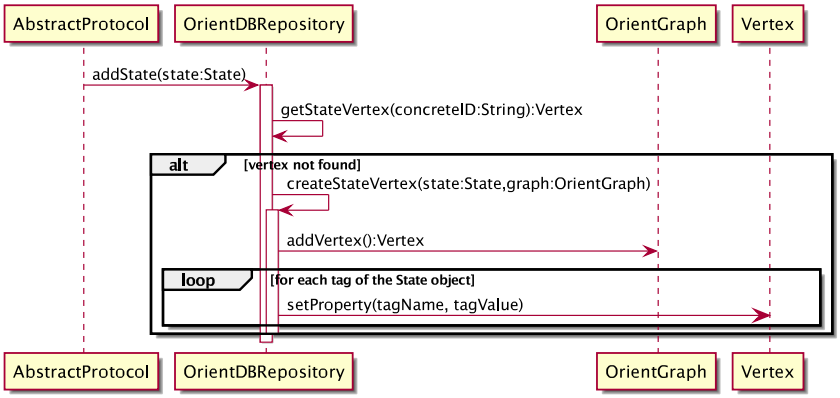


Figure 4.7: Sequence diagram demonstrating how a `State` is stored in the database

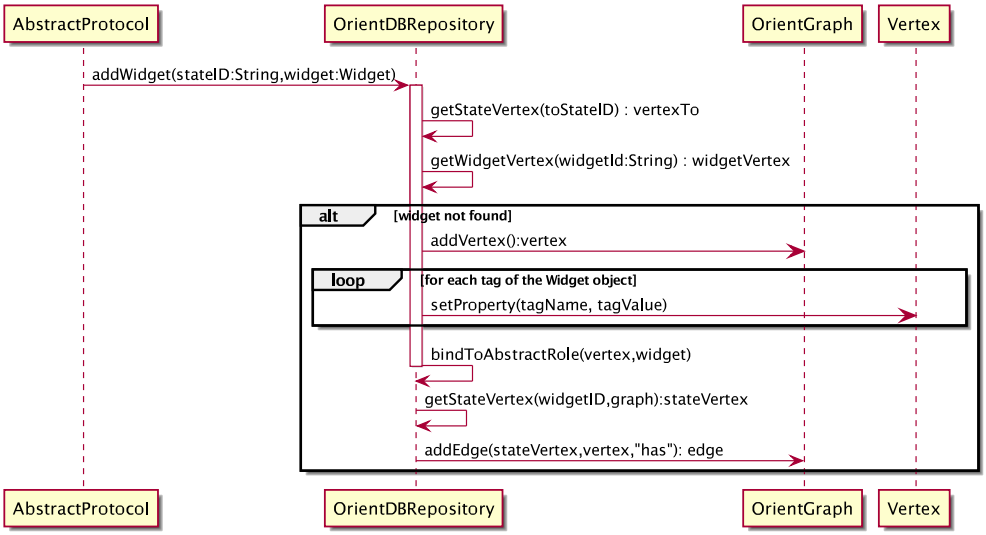


Figure 4.8: Sequence diagram demonstrating how a `Widget` is stored in the database

Figure 4.9 and 4.10 show how an `Action` is stored in the graph database. When an `Action` from a `Widget` to `State` is added, the from Vertex is retrieved using the "TargetID" property from the `Action`. When the `Action` is applied on a `State`, both the IDs for the `State` where the `Action` originates and the ID of the resulting `State` are provided.

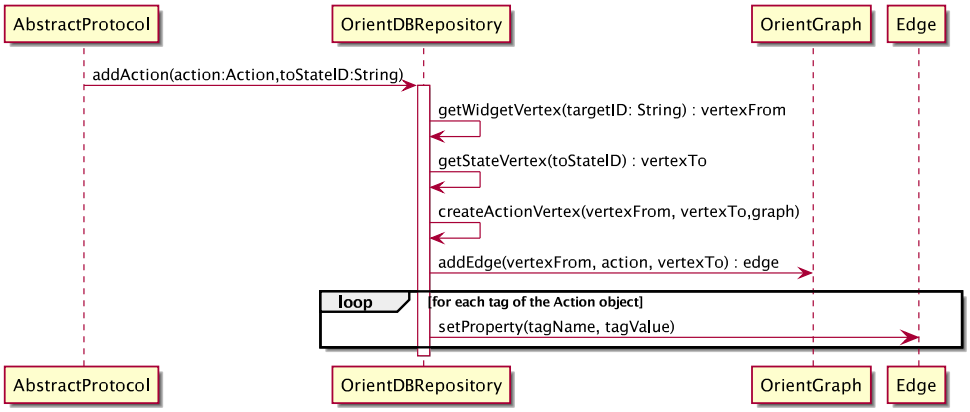


Figure 4.9: Sequence diagram demonstrating how an `Action` is stored in the database

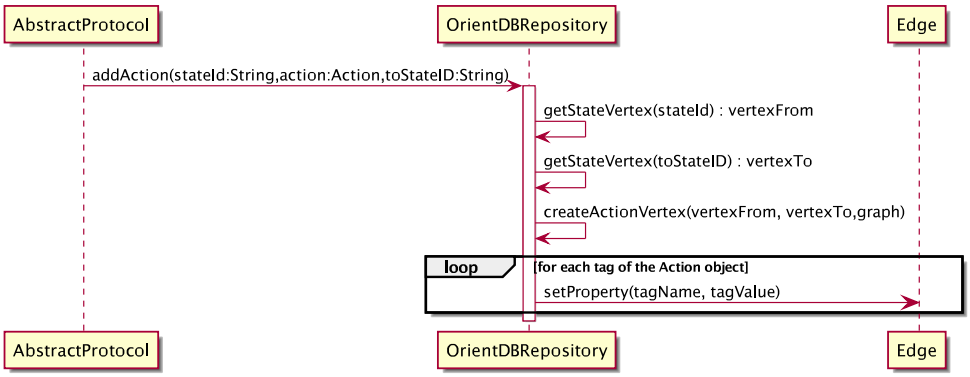


Figure 4.10: Sequence diagram demonstrating how an `Action` on `State` is stored in the database

5

EXTENDING THE DATA MODEL

In answer to Question Q2, we will investigate how the data model, presented in Section 4.2 and 4.3, can be extended. There are two ways to extend the data model that we describe in this Chapter:

1. Adding properties to an existing type (see Section 5.1).
2. Creating custom objects (see Section 5.2).

Both ways of extending the model allow the user to store the information required for their analyses using their own protocol implementation. We complete this chapter with an explanation on how data is stored for a given property and how the user can control it (see Section 5.3).

5.1. ADDING PROPERTIES

When an object is stored in the database, the storage method calls the `tags` method of `TaggableBase` (see Section 4.1.2) and stores each tag in the database as a property for the object taking the tag name as key and the value through the `toString` method of the type `T`. It is important to override the `toString` method if the stored type is not a simple type since Java will, by default, store a unique identification of the object. The JavaDoc of the class `Object` [28] describes the construction of this identification. Section 5.3 elaborates on this topic.
The best way to define custom tags in a protocol is to create a private class in the custom protocol which extends `TagBase`. In this class, the custom properties can be defined. Listing 5.1 provides an example.

Listing 5.1: Custom Tag definition

```
class CustomTags extends TagsBase {
    public static Tag<Long> ACTION_SEQUENCE = from("sequenceNumber",Long.class);
}
```

These custom tags can be applied in the protocol hooks to add a tag with a value to a `State`, `Action` or `Widget`. Listing 5.2 shows how a custom tag is added to the `Action` type at the time the `executeAction` method is called. This adds a sequence number to the executed `Action`.

Listing 5.2: Application of a custom Tag

```
@Override
protected boolean executeAction(SUT system, State state, Action action){
    action.set(CustomTags.ACTION_SEQUENCE,sequence++);
    return super.executeAction(system, state, action);
}
```

This section demonstrated how an object in the database can be extended with custom properties. This works fine for properties that are unique for specific objects. When there are properties that are shared between objects, it is better to store these properties in a custom object in the database. How this can be accomplished is explained in the next section.

5.2. CREATING CUSTOM OBJECTS

In some cases it is desirable to store a set of common properties for an `Action`, `State` or `Widget`. For instance, the `AbstractRole`, introduced in Section 4.2, has a relation to all `Action` vertices that share the same `AbstractRole`. Shared properties, like `AbstractState` for a State, `AbstractAction` for an Action and `AbstractRole`, `AbstractRoleTitle` and `AbstractRoleTitlePath` for a `Widget`, are already stored in the database.

When a user of TESTAR has a specific case that requires to store shared properties in the database, it is possible to define a `CustomType`, add the properties and store the `CustomType` in the database. In order to facilitate this feature, the `CustomType` is defined in the `graphdb` module. Figure 5.1 shows a class diagram for this type together with an example of a `CustomType` to store properties related to colour. Table 5.1 explains the properties for the `CustomType`.

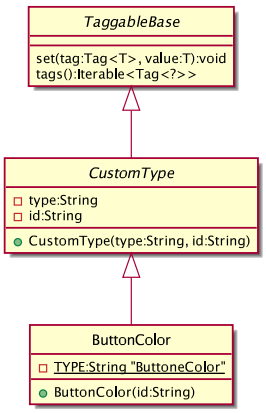


Figure 5.1: Class diagram showing the `CustomType` and a specific type (`ArtifactColor`)

Property	Description
type	Name for the type which is recorded in the database.
id	Identification of the instance.

Table 5.1: Description of the properties of the `CustomType`

Listing 5.3 shows an example of a type based on `CustomType`. `ArtifactColor` can be used to identify all values of a certain colour. Listing 5.4 shows how this type can be applied to link the button colour to an `Action`.

Listing 5.3: Custom type

```
import org.fruit.alayer.Tag;
import org.fruit.alayer.TagsBase;

public class ArtifactColor extends CustomType {
  private static final String TYPE = "ArtifactColor";
  public ArtifactColor(final String rgb) {
    super(TYPE,rgb);
  }
}

class ArtifactColorTags extends TagsBase {
  public static Tag<Integer> RED_VALUE = from("red", Integer.class);
  public static Tag<Integer> GREEN_VALUE = from("green", Integer.class);
  public static Tag<Integer> BLUE_VALUE = from("blue", Integer.class);
}
```

The class `ArtifactColorTags` defines available Tags (RED_VALUE, GREEN_VALUE and BLUE_VALUE). These tags are instantiated using the `from` method from the class `TagsBase`. Each Tag has an Integer type and are name “red” for the RED_VALUE Tag, “green” for the GREEN_VALUE tag and “blue” for the BLUE_VALUE tag. The class `ArtifactColor` has type `ArtifactColor` and the ID of the type is set from the field “rgb” in the constructor.
Listing 5.4 show the application of `ArtifactColor`. An instance identified by the string “ffffff” is created. Next the tags are set for each color.

Listing 5.4: Storing a custom type

```
@Override
protected boolean executeAction(SUT system, State state, Action action){
    ArtifactColor artifactColor = new ArtifactColor("ffffff");
    artifactColor.set(ArtifactColorTags.BLUE_VALUE,0xFF);
    artifactColor.set(ArtifactColorTags.RED_VALUE,0xFF);
    artifactColor.set(ArtifactColorTags.GREEN_VALUE,0xFF);

    graphDB.addCustomType(action, "coloured", artifactColor);
    action.set(CustomTags.ACTION_SEQUENCE, sequence++);
    return super.executeAction(system, state, action);
}
```

Extending TESTAR with the mechanism presented in this chapter makes the tool very flexible. Users can now construct their own data model on top of the default model provided by TESTAR. The only restriction is that the source for the relation with the `CustomType` needs to be available in the database. Figure 5.2 visualizes this idea.

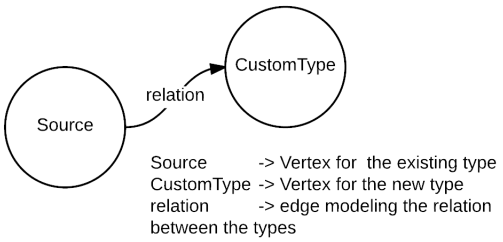


Figure 5.2: Schematic view of the position of a `CustomType` in the database

5.3. THE DATA TYPE OF THE EXTENSION

The module `graphdb` stores all tags for a type. To accomplish the task it uses the `toString` method of the type to retrieve the value. For simple types like `Boolean`, `Integer`, `Long`, `Double` and `String` this works fine. However, for complex types the result depends on the implementation of the `toString` method. By default the `toString` method of a complex type will result in a string representing the hash code of the object allowing the user to uniquely identify the object. By overriding the `toString` method, an appropriate format for the output can be created.

5.4. CONCLUSION

In answer to Question Q2, we can conclude that the design of TESTAR provides a lot of flexibility when it comes to adding extra information to the data model of TESTAR. To summarize, Table 5.2 lists the possibilities for the model extension.

Option	Reference
Add custom tags	Section 5.1
Add custom types	Section 5.2

Table 5.2: Options for the extension of the data model

Regarding the representation of the data, Section 5.3 explains how data is stored and how the user can control the output format. Which format is used will depend on the specific case.

6

Accessibility Standards for Automation

In our research, we explore accessibility evaluation of desktop software with an automated tool. To this end, we first have to select an accessibility standard to provide the rules that we can then use to define oracles and realize an implementation. This chapter deals with the first topic, selecting a standard.

In this chapter, we answer the following research question:
What is a suitable accessibility standard to evaluate the accessibility of desktop software? (Q6)
This question has the following sub-questions:

1. What is the current state of accessibility standards and which one is suitable? (Section 6.1)

2. What are the characteristics of the selected accessibility standard? (Section 6.2)

3. What can an automated tool accomplish with the rules in the selected accessibility standard? (section 6.3)

6.1. Accessibility Standards

Various countries and organizations have developed legislation, standards, norms or recommendations for the accessibility of digital products and services [31]. We collectively refer to these efforts as accessibility standards, even though not all are strictly called such. An accessibility standard will be a solid basis for accessibility evaluation.

6.1.1. REQUIREMENTS FOR THE ACCESSIBILITY STANDARD

Our research puts a number of requirements on the accessibility standard. A suitable standard will allow us to implement accessibility evaluation of desktop software in an automated tool and facilitates future research into the effectiveness of the implementation. To this end, the standard must be well-established, generic and applicable to desktop software. Moreover, at least some of the rules in the standard must be automatable. This section contains the requirements analysis and describes the standard that we selected.

From the previous text, we can extract the following requirements:

1. The standard is suitable for desktop software.
2. The standard contains automatable rules.
3. The standard is widely recognized.
4. The standard is not developed for one geographic area.
5. The standard is not bound to one group of disabilities.

Requirements 1 and 2 are a direct result of the research questions we formulated (see Section 1.1.2). The automation requirement does not exclude a standard if it also contains non-automatable rules. This relaxed definition was necessary to select an appropriate standard. The remaining requirements ensure that the selected standard is unbiased and forms a solid basis for evaluating accessibility. In this chapter we further discuss desktop software accessibility. Chapter 7 looks at automating accessibility rules.

6.1.2. CONTENT AND SOFTWARE ACCESSIBILITY

We can distinguish content accessibility, including web accessibility, and (desktop) software accessibility. In order to be accessible, content must have a semantic structure that can be used to present the content in a meaningful way. In contrast, software has interactive components that must be made accessible. The distinction is not a clear-cut line because content can be interactive. The web forms a good example. Besides static web pages there are dynamic, rich internet applications that look, feel and act like desktop software.

Content typically runs inside software. Accessible content in accessible software with appropriate assistive technologies yields an accessible experience. Assistive technologies are products that help users with disabilities to use a computer.

Our research requires an accessibility standard for software or, if we let go of the distinction between content and software, a standard for generic, interactive content. However, all of the reviewed standards target a specific type of content. Therefore, we selected a content standard that has supporting documentation explaining how to apply it to software.

There are standards for various types of content, but standards for the web are the most common. WCAG 2.0 [10] gives recommendations to make web content more accessible through user agents (software), such as web browsers and assistive technologies. WCAG 2.0 targets web content, not the user agents that present it. Likewise, Microsoft published information to make Office content more accessible [25]. The software, in this case the programs in the Microsoft Office suite, is made accessible by the vendor and takes advantage of accessible content such as documents, spreadsheets and e-mails. A last example is Portable Document Format (PDF) accessibility [**iso201614289**], known as PDF/Universal Accessibility (PDF/UA). Here, the PDF reader is the software that works with the accessible PDF content. The latter two standards do not satisfy Requirement 1, because they target specific content types and having no relation with generic desktop software. Among the reviewed standards, only WCAG 2.0 has such a relation.

Software includes user agents, text editors and document readers. There is also software where the line between software and content is less obvious, such as calculator programs. An accessibility standard for content is not usable unless it regards the software itself as content. This is what the supporting documentation of WCAG 2.0 makes possible. Therefore, we selected WCAG 2.0 as the standard to automate.

6.2. THE WCAG 2.0 ACCESSIBILITY STANDARD AND WCAG2ICT

WCAG 2.0 is a World Wide Web Consortium (W3C) Recommendation and has also been standardized [21]. In Europe, WCAG 2.0 is part of procurement standard EN 301 549 [11]. This makes WCAG 2.0 a well-established, international standard. It provides recommendations to increase accessibility for people with a wide range of disabilities. However, it is a standard for web content. We first discuss its general architecture and then elaborate on the adaptation to desktop software.

6.2.1. WEB CONTENT ACCESSIBILITY

Figure 6.1 depicts the components of WCAG 2.0 and how they relate to each other. WCAG 2.0 has a tree structure that comprises principles, guidelines and success criteria. There are four principles: Perceivable, Operable, Understandable and Robust ("POUR").

Each principle has one or more guidelines under it. Guidelines are basic goals that authors should work towards in order to increase content accessibility. Guidelines cannot be directly evaluated. For situations where evaluation is required, such as conformance certification, each guideline contains success criteria. For example, guideline 1.2 deals with alternatives for time-based media. Its success criteria provide requirements for pre-recorded audio-only or video-only content (1.2.1), captioning (1.2.2), audio descriptions (1.2.3) and more. The fact that a success criterion can be evaluated does not imply that it can also be automated, as we will demonstrate in Section 6.2.3.

Each success criterion has a conformance level. WCAG 2.0 distinguishes between three levels: A, AA and AAA. The conformance levels are cumulative, so conforming to level

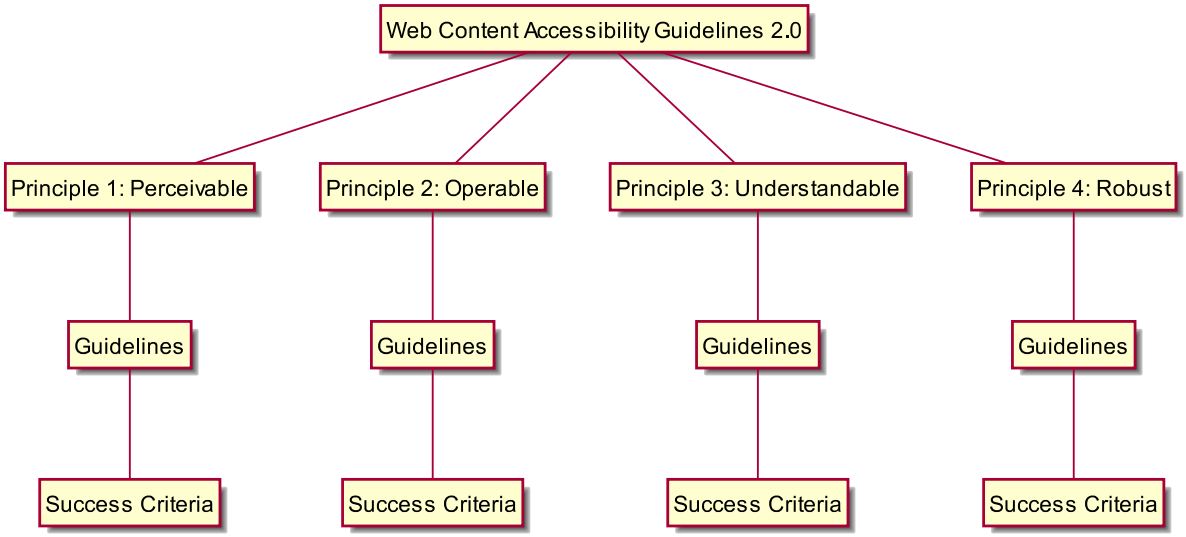


Figure 6.1: WCAG 2.0 structure with principles, guidelines and success criteria

AA requires passing more criteria than level A. Moreover, failing to meet criteria on level A is more serious than failing criteria on level AA, because the criteria on level A are more essential to a basic level of accessibility. Similarly, level AA is more important than level AAA.

The barrier to apply WCAG 2.0 to desktop software lies in the terminology used. WCAG 2.0 is built around web content and user agents to present that content. Reinterpreting these key terms leads to a suitable standard for evaluating desktop software accessibility.

6.2.2. DESKTOP SOFTWARE ACCESSIBILITY

WCAG 2.0 targets web content. However, the W3C Web Accessibility Initiative (WAI) provides guidance on how to apply WCAG 2.0 to non-web content as well, including software [23]. It is also known under the abbreviated name WCAG2ICT. WCAG2ICT specifically refers to non-web documents and software. The latter includes desktop software and thus satisfies Requirement 1. We will derive from this work in our research.

There are a few caveats to bare in mind when working with WCAG2ICT. WCAG2ICT contains only level A and AA success criteria, level AAA criteria are out of scope. It replaces web-related key terms with terms that apply in a non-web context. These replacements are informative rather than normative, so WCAG2ICT does not contain standardized criteria. Rather, it contains reinterpretations of the standardized criteria from WCAG 2.0. Furthermore, the authors of WCAG2ICT point out that it may not cover all accessibility requirements, because non-web content and software may have additional requirements that are irrelevant to web content and that are therefore not included in WCAG 2.0.

There are two key terms with significantly different interpretations in WCAG2ICT: content and user agent. The other key terms, as listed in the WCAG 2.0 glossary, remain largely unchanged. The two reinterpreted terms are:

Content Non-web content comprising documents and software. Documents require a user agent to present the content. For example, web pages and PDF files are documents. Software takes care of the content presentation by itself. For example, a calculator program is software. Software is the type we will be evaluating.

User agent Software that retrieves and presents documents. For example, web browsers and PDF readers are user agents. Software that incorporates its own content without retrieving it is not a user agent. Furthermore, software that provides a non-fully functioning presentation of the content, such as a preview, is not a user agent.

The terms document and software have more explicit definitions than the ones given here, but for our work they bare no significance. We do not elaborate on them further.

WCAG2ICT also introduces a new term: accessibility services of platform software refers to platform services that enable the exposure of information about user interfaces and events to software such as assistive technologies. This is typically realized through accessibility APIs.

With different interpretations for key WCAG 2.0 terms, we have a standard to evaluate the accessibility of desktop software. The next steps are interpreting the (automatable) success criteria and automating them. We conclude this section with an investigation into how straight-forward this interpretation is. In the next section we look at the role of automation.

6.2.3. THE ROLE OF EXPERTS

Evaluating success criteria in concrete situations, whether on the web or beyond, requires careful interpretation of not only the criteria, but also the content being evaluated. It would be incorrect to assume that all success criteria can be automated. We will show that this assumption is incorrect by giving a counter-example and looking at the role of experts in the evaluation process.

As an example, consider text alternatives for non-text content. Guideline 1.1 covers this topic. Success criterion 1.1.1 requires that all non-text content has a text alternative, except for special cases such as decorative content that should not be presented to assistive technologies. Success criterion 1.1.1 illustrates the challenges that experts face when evaluating WCAG 2.0 conformance of websites or software. They would have to decide if the image of a logo is purely decorative. Flagging such a special case as violating the criterion would lead to a false positive, but ignoring an image that is not a special case and that violates the criterion would lead to reduced accessibility. The situation becomes even more complicated when an automated tool performs the evaluation, since the tool cannot distinguish decorative images from functional ones. Therefore, the best it can do is raise a warning.

Other success criteria also warrant careful interpretation. A good illustration is a study in which twenty-two experts evaluated four web pages on all WCAG 2.0 success criteria [8]. 20% reported false positives, while 32% of the true problems went undetected. Of course,

this begs the question how the set of true problems can be guaranteed to be complete. Nevertheless, in general, expertise matters [9].

Fortunately, there are also automatable success criteria that are significant to accessibility. An example is success criterion 4.1.2. Guideline 4.1 describes technical compatibility. One of the requirements of success criterion 4.1.2 is that all user interface components have a programmatically determinable name and role. Assuming that the underlying accessibility API exposes this information, it is straight-forward to test for missing and/or incorrect values.

To summarize, we conclude that an automated tool can support experts performing accessibility evaluation, but such a tool cannot replace these experts. With adequate appreciation for the role of experts, WCAG2ICT contains the rules we need to automate accessibility evaluation. WCAG2ICT is suitable for desktop software and its guidance is based on WCAG 2.0, a set of well-established, generic guidelines containing both automatable and non-automatable rules. For the automatable rules, the question is what the automated tool can contribute to the evaluation. This is the topic of the next section.

6.3. THE ROLE OF AUTOMATED TOOLS

Desktop software can easily have many windows and the user interface components in these windows can have many different states. Experts will have a hard time evaluating the whole software, but automated tools can perform an extensive evaluation over night, although they too do not generally claim complete coverage. The topic of this section is what aspects of the accessibility evaluation process for desktop software can be automated, thereby more precisely defining the role of the automated tool.

In the broader field of automating usability evaluation of user interfaces, evaluation methods can be classified according to a four-dimensional framework [22]. The framework comprises the method class, method type, automation type and effort level. *Method class* is a high-level description of the evaluation conducted, such as simulation. *Method type* describes how the evaluation is conducted within the method class, such as processing information. *Automation type* is the evaluation aspect that is automated (see the rest of this section), such as capture. Finally, *effort level* describes the type (not the amount) of human effort required to execute the method, such as model development.

The *automation type* is the most revealing dimension for determining the role of automated tools. The framework distinguishes four types:

None There is no automation support.

Capture Data collection is automated.

Analysis Identification of potential problems is automated.

Critique Analysis and suggestions to resolve problems are automated.

Each automation type becomes more advanced and includes the type above it. That is, critique cannot be carried out without analysis and analysis cannot be carried out without capturing data. By definition, automated tools do not belong to the type *None*. The other types are all possible, though the exact type will depend on the specific tool.

The source of information for our oracles will be an accessibility API (see Section 7.1). Desktop software can be developed using a wide range of technologies. Accessibility APIs abstract these technologies to a generic representation. This means that the automated tool does not have access to concrete information about the technologies used to build the software being evaluated. This situation limits the options for critique, because useful critique does not describe what should be different in the representation from an API, but what should be changed in the specific technology.

At the other end, the minimum that an automated tool can do is collect data, the type *Capture*. Furthermore, with an accessibility standard containing automatable rules, the tool can perform analysis on that data. In the case of WCAG2ICT, the analysis would not be exhaustive, because not all success criteria are automatable. Experts can complete the analysis and also formulate suggestions to resolve problems, the type *Critique*. The automated tool in our research thus belongs to the *Analysis* type.

The automation type can be combined with the three other dimensions in the framework to obtain a full classification. The automated tool we will implement can then be classified as a guideline-based (method type) inspection tool (method class) for analysis (automation type) that requires minimal effort (effort level). Note that we will extend a tool that was not specifically designed to evaluate usability or accessibility problems, so there are other testing capabilities within the tool that fall outside this framework.

The classification in this section reveals that the analysis of the GUI, or more generally the desktop software through its GUI, is the aspect of the evaluation process that we can partially automate. Exactly how to do this and which guidelines to automate is the topic of Chapter 7.

6.4. CONCLUSION

Various accessibility standards exist, for a wide variety of areas. In our research, we investigate and implement accessibility evaluation of desktop software with an automated tool. The accessibility standard must form a solid basis for this research. We formulated requirements to this end.

WCAG2ICT, based on WCAG 2.0 was the only standard that satisfied the requirements. WCAG 2.0 has principles, guidelines and success criteria in a tree structure. A subset of the success criteria can be automated to some extent. Even when an automated tool is used, it can only complement the work of experts.

An automated tool can play different roles in accessibility evaluation. We strive to implement an analysis tool. This means that the tool does more than only capturing data,

but analysis does not imply giving suggestions for improvement based on the analysis. In our case, oracles are the way in which we perform analysis. We need to define these oracles before implementing accessibility evaluation in an automated tool.

7

ACCESSIBILITY ORACLES

Oracles in TESTAR define the information and rules that determine if a SUT passed a test. The WCAG2ICT accessibility standard we selected in Chapter 6 contains automatable accessibility rules. We can now investigate which rules can be automated with TESTAR and then define accessibility oracles for them. In Chapter 8 we will then implement these oracles.

In this chapter, we answer the following research question:
Which oracles can be defined for the selected accessibility standard? (Q7)
This question has the following sub-questions:

1. How does TESTAR retrieve information about the SUT? (Section 7.1)
2. What additional features become available for accessibility evaluation by incorporating a graph database? (Section 7.2)
3. What information needs to be in the evaluation report? (Section 7.3)
4. Which level A WCAG2ICT success criteria can be evaluated with oracles? (Section 7.4)
5. What are the limitations of accessibility oracles in TESTAR? (Section 7.5)

7.1. RETRIEVING INFORMATION FROM THE GRAPHICAL USER INTERFACE

An oracle applies a given rule, defined as a procedure, to actual and expected information from the SUT. TESTAR uses accessibility APIs to gather information about the SUT. These are the same APIs that assistive technologies, such as screen readers or magnification software, use to present the user interface to their users. In principle, this means that TESTAR has access to the same information as assistive technologies, although the latter

could employ additional techniques such as Optical Character Recognition (OCR). Our research focuses on TESTAR and hence on accessibility APIs, so this is the only technique we will investigate.

At the time of this research, TESTAR supports two accessibility APIs on Windows:

UIA UI Automation (UIA) is a modern, generic API for assistive technologies and automation tools.

JAB Java Access Bridge (JAB) is an API to make Java Swing software accessible.

Since our research focuses on the accessibility of Windows software in general, not on Java software in particular, we use UIA. Accessibility oracles will retrieve their information through this API.

UIA was already used to evaluate some accessibility rules with TESTAR [43], including images without a text alternative. There are also limitations resulting from the use of this API. TESTAR cannot establish the contrast ratio between foreground and background colours or the font size for text labels, because UIA does not expose colour and font information. Section 7.4 further discusses any API limitations we encountered, Section 7.5 contains a more general reflection on what we can expect from TESTAR.

7.2. EVALUATION TYPES

In this section, we describe the two evaluation types we used in TESTAR. These are the traditional, on-the-fly evaluation and the completely new, off-line evaluation.

Figure 7.1 shows the evaluation types. Traditionally, TESTAR performs on-the-fly evaluation in a test sequence, evaluating oracles for each state. Off-line evaluation becomes available with the integration of a graph database (see Chapter 3). The graph database stores test results and makes it possible to define oracles that use information from more than one state. Evaluation by an expert cannot be automated, but it is shown in the figure to reiterate the point that TESTAR serves as an automated tool in a larger context.

Each evaluation type corresponds to an evaluation step. The output from one step is input for the next. On-the-fly evaluation runs repeatedly on all the states TESTAR reaches during a test sequence. The state provides the information for on-the-fly evaluation and the results are stored in a graph database. Off-line evaluation runs after TESTAR finished a test sequence and uses the graph database to run queries. TESTAR can generate a report summarizing the results from both automated evaluation steps. This report serves as guidance for the expert, who ultimately gives an overall verdict. We now discuss the two automated evaluation steps in more detail.

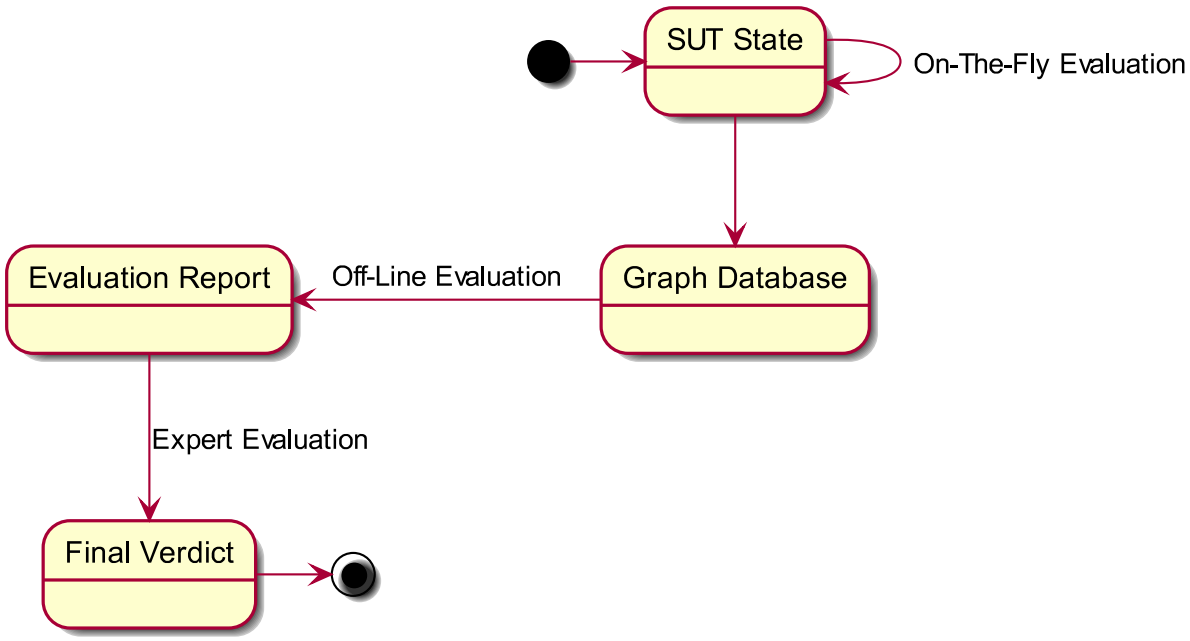


Figure 7.1: Accessibility evaluation types in TESTAR

7.2.1. ON-THE-FLY EVALUATION

TESTAR uses a widget tree to test the SUT.[1] A widget tree contains the widgets in a state, with parent and child relations to store the hierarchical structure. For example, a window can have a menu bar with menus that all have menu items, resulting in four layers in the hierarchy:

1. Window
2. Menu bar
3. Menu
4. Menu item

Every state in a test sequence is represented by a widget tree. The widget tree includes all the GUI components that the accessibility API exposes. Components that the API does not expose will not end up in the widget tree. These components cannot be discovered by TESTAR, although alternative methods to discover components could eventually be added to TESTAR. The components discovered through the accessibility API are the widgets in the widget tree. They can be evaluated according to a set of oracles. We refer to this type of evaluation as on-the-fly evaluation.

TESTAR performs two tasks during on-the-fly evaluation, both relevant to accessibility:

[1] Refer back to the TESTAR execution flow from Chapter 2 to see when TESTAR examines widgets.

1. Evaluate oracles
2. Derive actions

The two tasks are linked. For example, if we want to evaluate keyboard accessibility but TESTAR only derives mouse actions, the oracles cannot show whether or not the SUT is accessible through the keyboard. Therefore, oracles and actions are both part of this evaluation step. First, TESTAR examines the state to find violations. For example, TESTAR can find a low number of widgets with shortcut keys in the state. TESTAR then constructs the set of possible actions. For example, TESTAR can derive actions to simulate shortcut keys. To summarize, on-the-fly evaluation uses the state to evaluate oracles and derives actions to ensure that oracles lead to valid verdicts.

7.2.2. OFF-LINE EVALUATION

Before the integration of a graph database, TESTAR only performed on-the-fly evaluation. Using a graph database, off-line evaluation becomes possible. As the name indicates, this evaluation step takes place once TESTAR finished a test sequence and after the SUT shut down. The evaluation is off-line rather than on-the-fly because the SUT is no longer running. The term off-line is not related to network or internet connectivity.

A graph database makes it possible to consolidate results from on-the-fly evaluation and perform queries on the data. This allows us to evaluate oracles that use information from more than one state. For example, we could test if there exists a keyboard trap in the SUT.

A keyboard trap is a situation where it is possible to reach a user interface component via the keyboard, but not to leave it via the keyboard, thus creating a trap. The signs of a keyboard trap can be uncovered by only deriving keyboard actions and then querying the graph database to find a point at which focus stays on the same widget in every state. We refer to this type of evaluation as off-line evaluation. It facilitates additional oracles, but it is not a replacement for on-the-fly evaluation (see Section 7.5).

The two evaluation types complete our description of accessibility oracles. An oracle can be an on-the-fly oracle or an off-line oracle, corresponding to on-the-fly or off-line evaluation, respectively. An oracle yields a pass or a violation. A violation can be an error or a warning. An error is definite, while a warning requires verification by an expert (see Section 2.4). We can now define concrete accessibility oracles (see Section 7.4), but we will first define the contents of the evaluation report that will present the verdicts from TESTAR to the expert.

7.3. REPORTING EVALUATION RESULTS

Experts and TESTAR can both play a significant role in accessibility evaluation. The evaluation report is the link between them. It must provide all the information an expert needs

without becoming redundant or cluttered. In this section we define the information elements for such a report.

In addition to verifying verdicts from automated accessibility rules, an expert evaluates another, possibly overlapping set of rules. The evaluation process is more involved for non-automatable rules because the expert carries out all the work, but it should still lead to a number of verdicts. If we could accumulate these verdicts to obtain one metric to quantify the overall accessibility level, we could use it to compare the accessibility of different software. The evaluation report would need to contain the information required by the metric.

The topic of so-called accessibility metrics has been raised in various fields, including quality assurance and search engine optimization [39]. For the web, several accessibility metrics have been described [16, 17, 40, 32]. There appear to be no metrics for desktop software. However, this is not a problem as long as a metric is designed around WCAG 2.0, because we have the guidance from WCAG2ICT.

Web Accessibility Quantitative Metric (WAQM) [39] is the only accessibility metric in the reviewed literature to work with WCAG 2.0. It calculates a normalized accessibility rating based on errors, warnings and weights dependent on accessibility rule priority (conformance level). The rating is a reasonably reliable indicator of accessibility [40]. WAQM can work with different automated tools, but this requires tuning a number of variables [41]. Unfortunately, there are two problems that prevent us from gathering data for this metric with TESTAR.

First, WAQM requires the total number of evaluations as well as the number of problems per web page. However, WCAG2ICT splits the term web page into document and software (see Section 6.2). Software is a broader term than the single web page referred to in WAQM. There is a difference between evaluating a single web page, that tends to have a limited number of states, and evaluating software, that potentially has tens or hundreds of windows and associated states.

Second, the test methods of WAQM and TESTAR are incompatible. In on-the-fly evaluation, TESTAR evaluates the state of the software, executes an action to enter a new state and repeats the process. Thus, the software is evaluated multiple times. A web page is typically evaluated only once, then another web page is selected and evaluated. WAQM relies on the fact that there is one violation count for each web page. In contrast, TESTAR can evaluate an accessibility rule a fixed number of times yet get a different violation count each time, depending on the action selection algorithm being used. As WAQM uses a ratio of violation count to the total number of evaluations, the results TESTAR generates would have no meaning. The violation count will only be stable if a window in the software, not each state of that window, is evaluated once. Otherwise, the violation count could be lower or higher each time.

These two problems led to the decision to define our own evaluation report structure, independent of an accessibility metric. Analogous to the existing reports in TESTAR, results are grouped by state. The report has a slightly different structure depending on whether or

not graph database support is enabled, but the information elements are the same. With database support enabled, the report contains information only for the unique states. The information is first stored in the database and thereby filtered on uniqueness. The report is then generated from the information in the database. Without database support, the report contains information about each state from on-the-fly evaluation, off-line evaluation is obviously not performed. The information is written ad-hoc as soon as a state has been evaluated.

The report opens with general information:

- The report time.
- The report type, indicates if graph database support was enabled.
- The implementation version of the accessibility standard.
- The sequence number[2].
- The number of unique states with violations, when graph database support was enabled.

For each state, the evaluation report contains:

- The error count.
- The warning count.
- The pass count.
- The total number of evaluations performed on the state.
- General information about the state.
- A reference to a screenshot of the state.

For each error or warning, the evaluation report contains:

- A description.
- The widget, if applicable.
- The associated WCAG2ICT success criterion and its conformance level.
- A reference to further information from WCAG2ICT.

[2] If the `ForceToSequenceLength` setting is enabled, the sequence number in the evaluation report will be for the first sequence. However, the report could contain results from consecutive sequences if TESTAR had to start new sequences to reach the sequence length.

7.4. FROM ACCESSIBILITY RULES TO ACCESSIBILITY ORACLES

In Chapter 6, we selected an accessibility standard with automatable rules and in the previous section we defined two types of oracles to automate these rules. In this section, we discuss the oracles that we defined for accessibility evaluation of desktop software with TESTAR.

The oracles are based on WCAG 2.0 success criteria, because WCAG2ICT is the accessibility standard we selected. WCAG2ICT excludes level AAA success criteria.[3] Moreover, our research questions do not require an extensive implementation of all success criteria. Rather, they focus on the possibilities to automate accessibility evaluation of desktop software with TESTAR. For now, a proof-of-concept is a better choice than a fully-fledged solution. These considerations led to the decision to implement only level A success criteria. If the implementation proves useful in real-world scenarios (see Section 9.4), it can be extended with more rules, including the level AA success criteria in WCAG2ICT.

Success criteria typically have multiple requirements. An oracle typically automates one aspect of a success criterion, but with multiple oracles a success criterion can be automated fully. In the discussion that follows, it is important to keep in mind that the presence of an oracle does not imply that the associated success criterion is fully automated.

Guideline	Description	On-the-fly	Off-line
1.1	Text Alternatives	✓	
1.2	Time-based Media		
1.3	Adaptable		
1.4	Distinguishable		
2.1	Keyboard Accessible	✓	
2.2	Enough Time		
2.3	Seizures		
2.4	Navigable	✓	✓
3.1	Readable	✓	
3.2	Predictable		✓
3.3	Input Assistance		
4.1	Compatible	✓	

Table 7.1: WCAG 2.0 guidelines and oracle types

Table 7.1 summarizes the WCAG2ICT guidelines and the types of associated oracles we defined, if any. The table shows that our coverage is not complete.

Three limitations prevented us from defining oracles for the whole of WCAG2ICT:

1. The oracle would require actions from an expert.
2. The oracle would require information that is not available from the accessibility API.

[3] See Section 6.2.1 for the conformance levels.

3. Evaluating the oracle on any widget would trivially lead to a pass.

The rest of this section is organized by subsections for the guidelines with oracles. Each subsection begins with the guideline description from WCAG2ICT, followed by a table listing all level A success criteria in the guideline and an indication of which ones have oracles. We then provide the success criterion descriptions from WCAG2ICT. The oracle definitions conclude each subsection and form the most significant part. If not all success criteria in a guideline have oracles, we explain why. The last subsection discusses the guidelines that have no oracles at all.

We divided the oracle definitions into two parts: a brief overview of the oracle and a more in-depth description that also includes the limitations. The brief overview is structured as follows:

Criterion The number of the success criterion from WCAG2ICT.

Type The evaluation type, either on-the-fly or off-line.

Information The oracle information from the state and/or widget in on-the-fly oracles or the graph database in off-line oracles.

Procedure The oracle procedure that results in a pass or a violation.

Verdict The verdict if the oracle procedure does not result in a pass, either an error or a warning.

7.4.1. TEXT ALTERNATIVES

Guideline description [23]:

> Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, Braille, speech, symbols or simpler language.

Table 7.2 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
1.1.1	Non-text Content	✓	

Table 7.2: WCAG 2.0 success criteria in guideline Text Alternatives

Success Criterion 1.1.1: Non-text Content [23]:

> All non-text content that is presented to the user has a text alternative that serves the equivalent purpose, except for the situations listed below.

- Controls, Input: If non-text content is a control or accepts user input, then it has a name that describes its purpose. (Refer to Guideline 4.1 for additional requirements for controls and content that accepts user input.)
- Time-Based Media: If non-text content is time-based media, then text alternatives at least provide descriptive identification of the non-text content. (Refer to Guideline 1.2 for additional requirements for media.)
- Test: If non-text content is a test or exercise that would be invalid if presented in text, then text alternatives at least provide descriptive identification of the non-text content.
- Sensory: If non-text content is primarily intended to create a specific sensory experience, then text alternatives at least provide descriptive identification of the non-text content.
- CAPTCHA: If the purpose of non-text content is to confirm that content is being accessed by a person rather than a computer, then text alternatives that identify and describe the purpose of the non-text content are provided, and alternative forms of CAPTCHA using output modes for different types of sensory perception are provided to accommodate different disabilities.
- Decoration, Formatting, Invisible: If non-text content is pure decoration, is used only for visual formatting, or is not presented to users, then it is implemented in a way that it can be ignored by assistive technology.

ORACLES

Criterion 1.1.1

Type On-the-fly

Information Widget Role, Widget Title

Procedure Widget Role is Image and Widget Title is not empty

Verdict Error if the image is keyboard focusable, else Warning

This oracle tests for the presence of text alternatives on images. The text alternative comes from the title of the widget as reported by UIA. The oracle procedure treats an empty title as a missing text alternative.

There are actually two oracles, the verdict depends on whether or not the image is interactive. We define interactive as being able to receive keyboard focus (see Section 7.4.2). An interactive image that has no text alternative is an error, because without seeing the image it would be unclear what it does. A non-interactive image, such as a decorative image, without a text alternative is a warning, because it may not need a text alternative. Reporting this as error could cause false positives. However, that distinction is not made by WCAG2ICT, we only add it to lower the burden for the expert. The alternative solution would be to report all images as warnings, potentially leading to much more verification work.

7.4.2. KEYBOARD ACCESSIBLE

Guideline description [23]:

> Make all functionality available from a keyboard.

Table 7.3 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
2.1.1	Keyboard	✓	
2.1.2	No Keyboard Trap	✓	

Table 7.3: WCAG 2.0 success criteria in guideline Keyboard Accessible

Success Criterion 2.1.1: Keyboard [23]:

> All functionality of the content is operable through a keyboard interface without requiring specific timings for individual keystrokes, except where the underlying function requires input that depends on the path of the user's movement and not just the endpoints.
> Note 1: This exception relates to the underlying function, not the input technique. For example, if using handwriting to enter text, the input technique (handwriting) requires path-dependent input but the underlying function (text input) does not.
> Note 2: This does not forbid and should not discourage providing mouse input or other input methods in addition to keyboard operation.

Success Criterion 2.1.2: No Keyboard Trap [23]:

> If keyboard focus can be moved to a component of the non-web document or software using a keyboard interface, then focus can be moved away from that component using only a keyboard interface, and, if it requires more than unmodified arrow or tab keys or other standard exit methods, the user is advised of the method for moving focus away.
> Note: Since any content that does not meet this success criterion can interfere with a user's ability to use the whole non-web document or software, all content on the non-web document or software (whether it is used to meet other success criteria or not) must meet this success criterion.
> Note: Standard exit methods may vary by platform. For example, on many desktop platforms, the Escape key is a standard method for exiting.

ORACLES

Criterion 2.1.1

Type On-the-fly

Information Widget ShortcutKey, Threshold

Procedure total Widget ShortcutKeys in State is greater than Threshold

Verdict Warning

This oracle tests how many widgets in a state have shortcut keys. On Windows with UIA, these can be either access keys or accelerator keys. An access key is a letter of the widget title that activates that widget when pressed along with a modifier key, usually `Alt`. An example is `Alt+F` to open the File menu. An accelerator key directly executes a command without the need for a specific widget to be visible. An example is `Ctrl+P` to print. If the ratio of widgets with shortcut keys and those without shortcut keys drops below a predefined threshold, this could indicate poor keyboard accessibility. The threshold is currently defined in the oracle procedure with a value requiring that at least one in three widgets has a shortcut key. An expert must still verify if the whole SUT is fully keyboard accessible (Problem 1).

To support evaluating oracles for this guideline, TESTAR also derives actions. The set of actions comprises standard keyboard actions that are likely to work in most circumstances, such as pressing the `Tab` key, and actions based on the shortcut keys found in the SUT. However, shortcut keys are not strictly standardized. Instead of `Ctrl+P`, the values `Ctrl-P` or `Control+P` would also be valid. Moreover, a SUT does not have to expose a key through UIA to make it function. Therefore, we made this oracle and the next one a warning and not an error.

Criterion 2.1.2

Type On-the-fly

Information Widget ConcreteID, Widget HasKeyboardFocus, Threshold

Procedure Widget ConcreteID for Widget with HasKeyboardFocus changes within number of consecutive States less than Threshold

Verdict Warning

This oracle tests for simple keyboard traps. If the same widget has keyboard focus in a number of consecutive states exceeding a predefined threshold, this could indicate a keyboard trap (see Section 7.2.2). The threshold is currently defined in the oracle procedure with a value requiring that the same widget does not have focus more than five times in a row. The uniqueness of a widget is determined by its "ConcreteID" property (see Section 4.1).

An important property of this oracle is that it tests the widget ID, not the state ID. The corresponding success criterion defines a keyboard trap within a user interface component. In other words, the criterion is not violated if focus stays within the same state, but

only if it is trapped in one component of that state. It is possible that keyboard focus gets trapped inside a sub-window with multiple widgets, such as a set of search controls within a larger window, but this oracle only tests if keyboard focus stays on the same widget, hence why we spoke of simple keyboard traps.

The success criterion only applies if keyboard focus can be placed in a component via the keyboard. An example of another way to move keyboard focus that does not qualify is clicking the mouse to move the cursor. Since TESTAR only executes keyboard actions during accessibility evaluation, this condition is always met, but there is a catch. TESTAR does not necessarily try all available shortcut keys. On top of that, according to the success criterion, the user should be advised of non-standard keys to leave a component. TESTAR cannot interpret such advice. Work for an expert remains here (Problem 1).

7.4.3. NAVIGABLE

Guideline description [23]:

> Provide ways to help users navigate, find content, and determine where they are.

Table 7.4 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
2.4.1	Bypass Blocks		
2.4.2	Page Titled		✓
2.4.3	Focus Order		
2.4.4	Link Purpose (In Context)		

Table 7.4: WCAG 2.0 success criteria in guideline Navigable

Success Criterion 2.4.1: [23]:

> (for non-web documents)
> A mechanism is available to bypass blocks of content that are repeated on multiple non-web documents in a set of non-web documents.
> (for software programs)
> A mechanism is available to bypass blocks of content that are repeated on multiple software programs in a set of software programs.

Success Criterion 2.4.2: Page Titled [23]:

> Non-web documents or software have titles that describe topic or purpose.

Success Criterion 2.4.3: Focus Order [23]:

> If non-web documents or software can be navigated sequentially and the navigation sequences affect meaning or operation, focusable components receive focus in an order that preserves meaning and operability.

Success Criterion 2.4.4: Link Purpose (In Context) [23]:

> The purpose of each link can be determined from the link text alone or from the link text together with its programmatically determined link context, except where the purpose of the link would be ambiguous to users in general.

ORACLES

Criterion 2.4.2

Type Off-line

Information Widget Role, Widget Title, Threshold

Procedure Widget Role is Window and Widget Title occurs in number of States less than Threshold

Verdict Warning

This oracle tests for ambiguous window titles. Software should have a title that describes topic or purpose. According to WCAG2ICT, the name of the software itself is a sufficient title if it describes the topic or purpose. In complex software, such as an advanced word processor, the window title probably does not adequately describe the topic or purpose if it only contains the name of the software, even though it passes the success criterion, so we give the expert the opportunity to spot suspicious window titles. This oracle cannot test if the titles describe the topic or purpose, since that requires interpretation from an expert (Problem 1), but it can find duplicate titles and warn the expert about them.

In web content, each web page should have a title that describes topic or purpose. In software, if a number of windows greater than a predefined threshold have the same title, this could mean that the SUT contains generic titles that do not describe the topic or purpose. The threshold is currently defined in the oracle procedure with a value requiring that no more than three windows have the same title. However, multiple windows with the same title are not necessarily an error. For example, there could be multiple message box windows with the same title. That is why the oracle yields a warning.

SUCCESS CRITERIA WITHOUT ORACLES

Success criterion 2.4.1 trivially passes (Problem 3). TESTAR evaluates a program in isolation, without considering results from other programs. This success criterion always passes in such cases, because it applies to software programs in a set of software programs, as

WCAG2ICT also points out. Furthermore, WCAG2ICT notes that sets of software programs that meet the definition in this success criterion appear to be extremely rare.

Success criterion 2.4.3 requires expertise (Problem 1). TESTAR knows what widget has focus and can change the focus, but it does not know if the navigation sequences affect meaning or operation. TESTAR also cannot test that the order preserves meaning and operability.

Success criterion 2.4.4 also requires expertise (Problem 1). UIA has a role to denote (hypertext) links. The link name and its context can be programmatically determined by TESTAR, but the link purpose cannot.

7.4.4. READABLE

Guideline description [23]:

> Make text content readable and understandable.

Table 7.5 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
3.1.1	Language of Page	✓	

Table 7.5: WCAG 2.0 success criteria in guideline Readable

Success Criterion 3.1.1: Language of Page [23]:

> The default human language of non-web documents or software can be programmatically determined.

ORACLES

Criterion 3.1.1

Type On-the-fly

Information Widget Role, Widget Locale

Procedure Widget Role is Window and Widget Locale is not 0

Verdict Error

This oracle tests for missing language identifiers. UIA exposes the locale of all widgets, but the documentation [26] states that it is typically set on windows rather than individual user

interface components. The oracle procedure uses the locale on a window to determine the language and treats it as missing if the locale is equal to 0.

There is one case where reporting a missing language identifier as error is incorrect, namely when the SUT follows the language configured in the operating system. In that case, a missing language identifier could mean that the default language is in use. The oracle cannot test if the programmatically determined language matches the actual language of the text in the SUT. This would require a text analysis API (Problem 2) or an expert (Problem 1).

7.4.5. PREDICTABLE

Guideline description [23]:

> Make non-web documents or software appear and operate in predictable ways.

Table 7.6 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
3.2.1	On Focus		✓
3.2.2	On Input		

Table 7.6: WCAG 2.0 success criteria in guideline Predictable

Success Criterion 3.2.1: On Focus [23]:

> When any component receives focus, it does not initiate a change of context.

Success Criterion 3.2.2: On Input [23]:

> Changing the setting of any user interface component does not automatically cause a change of context unless the user has been advised of the behaviour before using the component.

ORACLES

Criterion 3.2.1

Type Off-line

Information Action Type, State ConcreteID

Procedure Action Type is InWindow and old State ConcreteID is new State ConcreteID

Verdict Error

This oracle tests for unexpected state changes resulting from keyboard input. When a user interface component receives focus, it should not initiate a state change. The oracle tests that when a component receives focus, it does not change the state to one with a different "ConcreteID" property (see Section 4.1). This is not a complete test, because upon receiving focus a widget could also move focus to another widget without changing the state in such a way that TESTAR generates a different ID. TESTAR cannot test this. By the time it examines the new state, the focus change already happened. In reverse, a state change can cause a focus change, for example when a new window opens and a different widget receives focus. Such behaviour is not a violation.

The oracle queries the graph database for actions that should stay within the same window. To this end, we set a tag on these actions to mark them as navigating in the same window. On Windows, the actions are `Tab` and `Shift+Tab`. The other keys TESTAR tries may navigate to a different window. For example, the arrow keys typically move the cursor through text or menus and thereby stay in the same window, but they can also change to a new window if the widget they are used on triggers a window update. Tab controls are an example where each tab that is selected using the arrow keys updates the window.

Next, the oracle filters the actions on the IDs of the starting and the resulting state. If the IDs are different, there was an unexpected state change. The oracle yields an error containing the window title from the resulting state.

SUCCESS CRITERIA WITHOUT ORACLES

Success criterion 3.2.2 requires expertise (Problem 1). It is similar to 3.2.1, but describes changing the settings of a user interface component. An example is a date picker with selection lists for year, month and day. Selecting the month could update the day list to reflect the number of days in the selected month. We did not define oracles because software does not violate this success criterion if the user has been advised of the behaviour before using the component, requiring evaluation by an expert.

A pitfall for success criterion 3.2.2 is that there are often multiple ways to change the settings of a component. All of these have to be considered. For example, a check-box can be checked with `Space`, but possibly also by using a shortcut key. Instead of observing what happens if any of these methods is used, a better solution is to listen for change events on the component. By design, this is not how TESTAR works.

7.4.6. COMPATIBLE

Guideline description [23]:

> Maximize compatibility with current and future assistive technologies and

accessibility features of software.

Table 7.7 presents all level A success criteria in this guideline.

Criterion	Description	On-the-fly	Off-line
4.1.1	Parsing		
4.1.2	Name, Role, Value	✓	

Table 7.7: WCAG 2.0 success criteria in guideline Compatible

Success Criterion 4.1.1: Parsing [23]:

> For non-web documents or software that use mark-up languages, in such a way that the mark-up is separately exposed and available to assistive technologies and accessibility features of software or to a user-selectable user agent, elements have complete start and end tags, elements are nested according to their specifications, elements do not contain duplicate attributes, and any Ids are unique, except where the specifications allow these features.
> Note: Start and end tags that are missing a critical character in their formation, such as a closing angle bracket or a mismatched attribute value quotation mark are not complete.

Success Criterion 4.1.2: Name, Role, Value [23]:

> For all user interface components (including but not limited to: form elements, links and components generated by scripts), the name and role can be programmatically determined; states, properties, and values that can be set by the user can be programmatically set; and notification of changes to these items is available to user agents, including assistive technologies.
> Note: This success criterion is primarily for software developers who develop or use custom user interface components. Standard user interface components on most accessibility-supported platforms already meet this success criterion when used according to specification.

ORACLES

Criterion 4.1.2

Type On-the-fly

Information Widget Role, Widget Title

Procedure Widget Role is not Image and Widget Title is not empty

Verdict Error

This oracle tests for empty widget names. Trivially, UIA exposes the name, role and value of all user interface components, thereby allowing them to be programmatically determined (Problem 3). Because an empty name or unknown role affects both TESTAR and assistive technologies, we regard them as errors. This means that the scope is broadened from the success criterion, because technically the empty name can be programmatically determined so there is no violation. However, an empty name is not meaningful, because it does not provide any additional information about the user interface component. Because the oracle for success criterion 1.1.1 already tests the same for images, we exclude images here to avoid duplicate violations.

Criterion 4.1.2

Type On-the-fly

Information Widget Role

Procedure Widget Role is not Unknown

Verdict Error

This oracle tests for unknown widget roles. In UIA, `UIAUnknown` is the only unknown role. The other UIA roles all represent a specific type of user interface component. The remarks about programmatically determining information from the previous oracle (Problem 3) apply to this one as well.

Success criterion 4.1.2 has more requirements, including that states, properties and values that can be set by the user can also be set programmatically. UIA allows to do this for standard user interface components (Problem 3). Non-standard components may not provide this functionality and in turn TESTAR will fail to fully evaluate the SUT (Problem 2). While this rule has no oracle, the lack of evaluation results would still point to a potential problem. An example was evaluating Java Swing software before TESTAR gained support for the JAB API.

SUCCESS CRITERIA WITHOUT ORACLES

Success criterion 4.1.1 trivially passes (Problem 3). If content cannot be parsed, TESTAR cannot test it. Syntactically checking the content for parsing problems needs to be carried out in a step prior to evaluation with TESTAR. If TESTAR could test the content, it passed this criterion. Here we see an analogy with missing functionality in an accessibility API, which TESTAR can also not test directly.

7.4.7. GUIDELINES WITHOUT ORACLES

A number of guidelines do not have any oracles. An explanation follows. Section 7.5 further reflects on the process to define oracles and the limitations we encountered.

Guideline 1.2: Time-based Media [23]:

> Provide alternatives for time-based media.

This guideline requires expertise (Problem 1) because the information must form an equivalent alternative for the content. Determining this equivalence requires knowledge of the content and the alternative. Moreover, the presence of the alternative cannot be automatically detected, because there is nothing to programmatically distinguish it from other content.

Guideline 1.3: Adaptable [23]:

> Create content that can be presented in different ways (for example simpler layout) without losing information or structure.

This guideline requires expertise (Problem 1) because the meaning and characteristics of information and the structural relationships must be determined. TESTAR can show the structure it can programmatically determine, but an expert is needed to link this to the actual information and structure.

Guideline 1.4: Distinguishable [23]:

> Make it easier for users to see and hear content including separating foreground from background.

This guideline requires expertise (Problem 1) because it must be possible to control audio that automatically plays for more than 3 seconds. The presence of audio controls cannot be tested by TESTAR without specific configuration. It must be possible to pause or stop audio or to change its volume. There is nothing to programmatically distinguish such controls from other user interface components.

This guideline also requires information from the accessibility API that is not available (Problem 2). Colour must not be the only means to distinguish information. Besides the fact that TESTAR cannot identify the other means (Problem 1), the accessibility API we use also does not expose colour information.

Guideline 2.2: Enough Time [23]:

> Provide users enough time to read and use content.

This guideline requires expertise (Problem 1) because the amount of time and the way in which it should be provided depends on the content. Some content can be paused or hidden, while other content has adjustable time limits. TESTAR cannot determine these requirements without specific configuration.

Guideline 2.3: Seizures [23]:

> Do not design content in a way that is known to cause seizures.

This guideline requires expertise (Problem 1) because the cause for seizures, anything that flashes rapidly or in a particular way, cannot be tested with the types of oracles we have at our disposal. Flashes can only be seen by observing a window for a period of time to see if it changes and by design, that is not how TESTAR works.

Guideline 3.3: Input Assistance [23]:

> Help users avoid and correct mistakes.

This guideline requires expertise (Problem 1) because the help consists of text descriptions, labels or instructions. TESTAR has access to the content, but cannot determine if it is adequate or actually helpful.

7.5. REFLECTING ON THE ORACLES

Initially, it appeared possible to define oracles for roughly 50% of the level A success criteria in WCAG2ICT. When defining oracles, this estimate proved to be too high. Looking at the previous section, we defined oracles for seven out of the twenty-five level A success criteria, leading to 28% having oracles. A further nuance is that the existence of an oracle for a success criterion does not imply full automated test coverage of that criterion. Success criterion 4.1.2 is a good example of this. Overall, the oracle coverage confirms that expertise matters (see Section 6.2.3).

One difficulty when defining oracles is that many success criteria require that all aspects of the SUT meet a certain requirement. In such cases, TESTAR cannot prove the absence of violations, because TESTAR does not guarantee full test coverage for the SUT. This problem is two-fold. Take the example of keyboard accessibility. First, even if no violations are detected, it is not generally possible to guarantee that TESTAR visited all states, so unexplored parts of the SUT may still contain violations. Second, if TESTAR fails to reach a widget via the keyboard and signals a violation, it is not guaranteed that there is no other way to reach that widget using the keyboard. An extreme example that would report poor keyboard accessibility is a test sequence with only No Operation (NOP) actions.

Another difficulty lies in the need to define oracles specific to the SUT. For example, consider the success criteria for making content more distinguishable (guideline 1.4). Among other requirements, it must be possible to pause, stop or control the volume of audio that plays automatically for more than 3 seconds (success criterion 1.4.2). Finding a pause button or volume slider in a given state can be done with TESTAR, but only if the oracle contains information to identify that button or slider. A slider could do many other things not related to changing the volume. So more oracles could be defined with knowledge of a specific SUT, but the generic ones we defined should still cover a broad range of violations.

7.6. CONCLUSION

TESTAR interacts with the SUT through its GUI. The information about the GUI comes from various accessibility APIs. We use UIA to evaluate the accessibility of desktop software on Windows.

Traditionally, TESTAR performs on-the-fly evaluation. With the integration of a graph database, we can also perform off-line evaluation. This makes it possible to define oracles that span multiple states.

TESTAR will communicate the evaluation results through an evaluation report. The report does not aim to provide all the information needed to compute an accessibility metric. Rather, the report contains the most important details about each violation and also includes general information. The selection of details to include is a balance between thoroughness and readability.

Not all requirements of all success criteria can be automated with oracles. In fact, success criteria that can be automated can generally not be covered completely by oracles. Even with the automation from TESTAR, experts play an important role in accessibility evaluation. Although we took this consideration into account when estimating how many oracles we could define, the actual number is still somewhat underwhelming.

A limitation of our approach is that we lean towards the side of caution when it is not completely certain that a violation is an error. The result is more oracles that yield warnings than ones that yield errors. As warnings require verification, this means more work for an expert. If we were to also define oracles for a specific SUT, we could evaluate more aspects of success criteria. Instead, we have generic oracles that can be implemented with information from the accessibility API.

8

Implementing Accessibility Evaluation in TESTAR

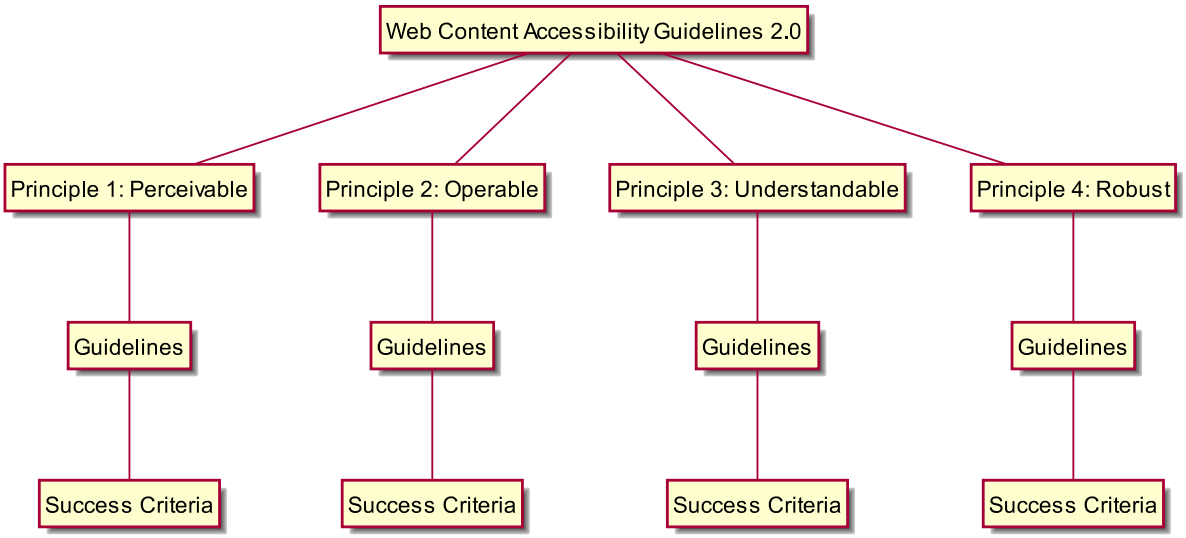


Figure 8.1: WCAG 2.0 structure with principles, guidelines and success criteria

WCAG 2.0, the basis for WCAG2ICT, comprises principles, guidelines and success criteria. Figure 8.1 shows this structure again. It is the same figure from Section 6.2. An implementation of accessibility evaluation in TESTAR can follow this structure, but must also be generic enough to allow exchanging WCAG2ICT for another set of rules. The implementation we present in this chapter follows a design to make that possible.

In this chapter, we answer the following research question:
How can TESTAR be extended to support accessibility evaluation? (Q8)

8.1. OVERVIEW

Our design extends TESTAR with new test protocols as well as classes to implement accessibility standards, evaluation results and utilities. The new test protocols coordinate accessibility evaluation with a particular standard. They are similar to other test protocols in TESTAR, such as the generic desktop protocol. A protocol uses the implementation of an accessibility standard to perform on-the-fly and off-line evaluation and to derive actions. A protocol receives evaluation results through classes that implement single results and a collection of results. Utilities provide supporting functionality.

Few parts of TESTAR had to be modified to support accessibility evaluation. In the core module, we added the possibility to compile actions that press shortcut keys to `StdActionCompiler` and `AnnotatingActionCompiler`. We also added unit tests to this module and ran the code when validating our implementation. This led to resolving some minor problems in other classes. In the `windows` module, we updated `UIATags` to support more UIA properties and also retrieve them from the API. Except for these modifications, the existing code remains unmodified.

Our implementation interacts with the existing code of TESTAR in several places:

- `DefaultProtocol` to extend into an accessibility protocol.
- `State` and `Widget` during on-the-fly evaluation.
- `Verdict` to return a concise evaluation result.
- `StdActionCompiler` to compile the derived actions.

Figure 8.2 shows an example of interactions between the most important classes of our implementation. The figure visualizes the on-the-fly evaluation process. The protocol calls an accessibility standard, in this case the implementation of WCAG2ICT. Our implementation recursively calls the accessibility standard as a whole, then the principles and finally the guidelines, but this is not a requirement. The accumulated evaluation results are returned to the protocol. The following sections discuss the implementation in the order of the figure. Not shown in the figure are the utility classes, that we discuss last.

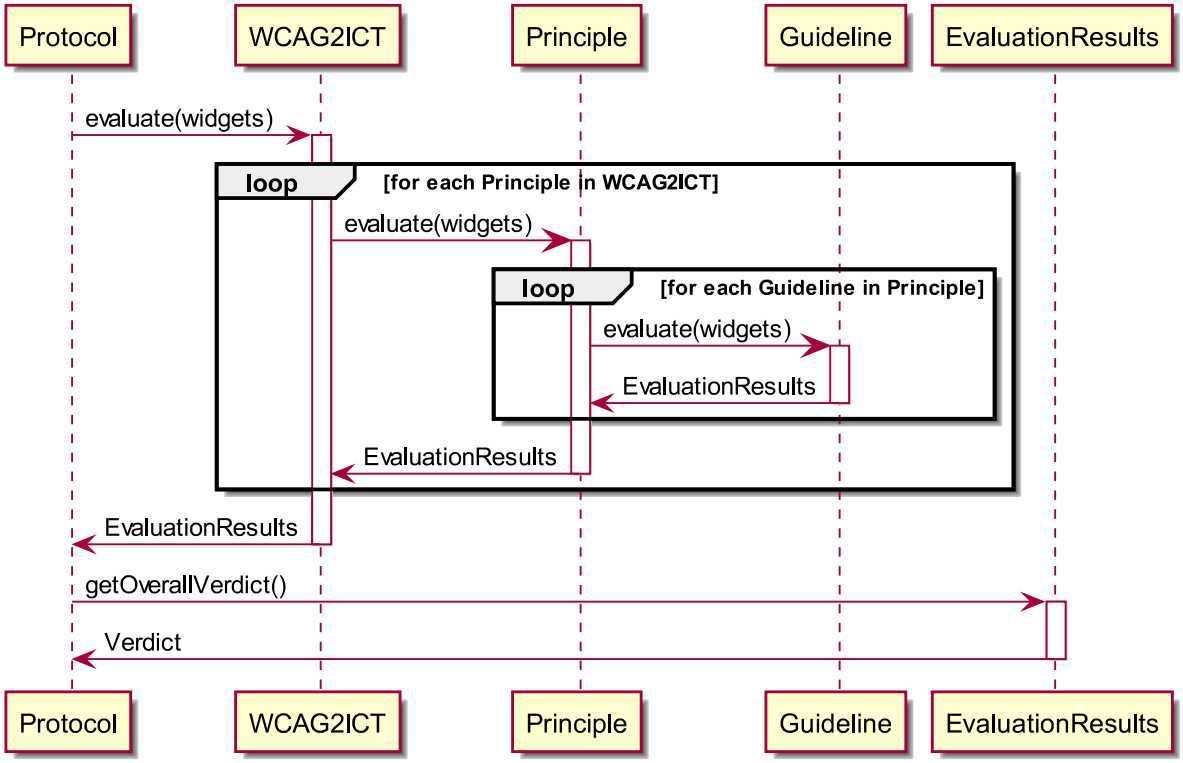


Figure 8.2: Sequence diagram demonstrating the recursive implementation of on-the-fly accessibility evaluation

8.2. TEST PROTOCOLS

Figure 8.3 displays the WCAG2ICT accessibility protocol and its super-classes. The responsibilities of the protocol classes are as follows:

AbstractProtocol Specifies protocol methods and implements basic functionality.

DefaultProtocol Implements a default, generic protocol with the most essential oracles.

AccessibilityProtocol Specifies additional methods for accessibility evaluation and provides a reference implementation. It outputs an evaluation report through the `HTMLReporter`.

Protocol_accessibility_wcag2ict Is a wrapper for `AccessibilityProtocol` with a concrete `Evaluator`, namely `WCAG2ICT`. It can be used as a protocol class from a TESTAR settings file.

`AccessibilityProtocol` extends `beginSequence` to open the evaluation report and extends `finishSequence` to close it. When graph database support is enabled, `finishSequence` also starts off-line evaluation by calling `offlineEvaluation` before closing the evaluation report. `AccessibilityProtocol` also coordinates the actual evaluation process.

`AccessibilityProtocol` extends `getVerdict` and `deriveActions` to evalaute on-the-fly oracles and derive actions, respectively. `getVerdict` stores `EvaluationResults`

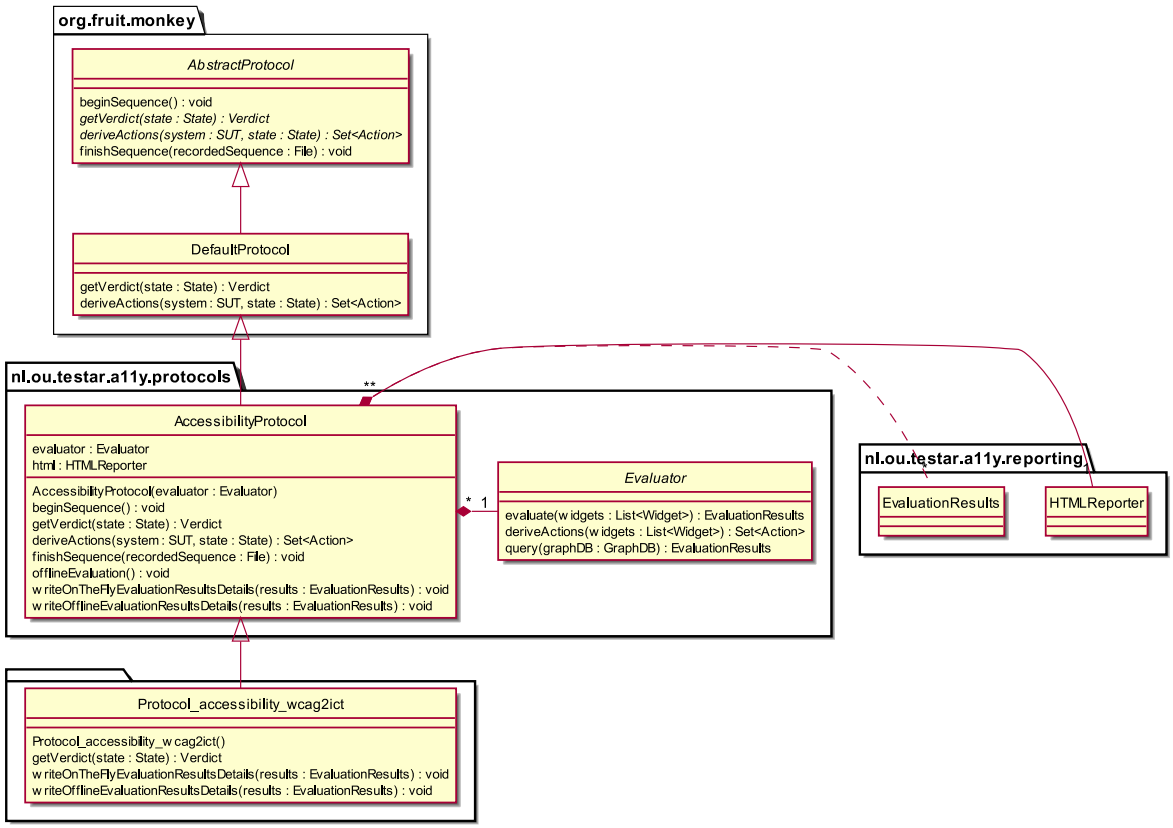


Figure 8.3: Class diagram showing the accessibility protocol classes

and the number of errors, warnings and passes as tags on the evaluated `State`. The WCAG2ICT protocol further extends `getVerdict` to set a tag containing success criterion violations in a machine-readable format on the `State`.

`deriveActions` first calls the super-class to derive actions using the default protocol. An example of such an action is the one to bring the SUT back into the foreground when needed. If there are any actions, the method returns them. Otherwise, it derives and returns accessibility actions.

`AccessibilityProtocol` makes calls on an `Evaluator` to perform on-the-fly evaluation, derive actions and query a graph database in off-line evaluation. `Evaluator` is an interface that a class must implement in order to be called from an accessibility protocol. In the WCAG2ICT protocol, the `Evaluator` is the `WCAG2ICT` class that contains WCAG2ICT oracles (see Section 8.3). It is possible to evaluate other rules by providing a different `Evaluator` implementation.

The `Evaluator` interface has three methods. Method `evaluate` performs on-the-fly evaluation. It takes a list of relevant widgets and returns `EvaluationResults` (see Section 8.4). Relevant widgets are those widgets that the protocol wants to evaluate. Accessibility standard classes can further narrow down the list of widgets that they use in oracles, for example to look for missing text alternatives only on images, but the protocol always offers the same list. This list can be shorter than the list of all widgets in a state. In the WCAG2ICT protocol, the list contains all the widgets in the state, except:

- Widgets that do not pass the black-list and white-list filtering engines in TESTAR.
- Widgets that are not the top-most widgets as determined by their Z-index (see Section 4.1).
- Decorational and disabled widgets.
- The `State` itself, which is the root of the widget tree.

Method `deriveActions` derives accessibility actions. It also takes a list of relevant widgets and returns a set of actions. In the WCAG2ICT protocol, the list of widgets is the same as the one used in on-the-fly evaluation, but this does not generally have to be the case.

Method `query` queries a graph database for off-line evaluation. It takes a reference to a graph database (see Section 4.3) and returns `EvaluationResults` similar to `evaluate`. The queries can be formulated in Gremlin (see Section 3.4.2) and each query is sent to the database separately. In our implementation, the method body contains the queries as strings, but other options are possible. For example, the queries could reside in the file system. There are also alternatives for Gremlin, but we did not implement them in the `graphdb` module.

Evaluation results are reported in an evaluation report (see Section 7.3). To write the report, there is a utility class `HTMLReporter` (see Section 8.5). The implementation-specific details, such as information about success criteria in the WCAG2ICT protocol, are written

by two methods: `writeOnTheFlyEvaluationResultsDetails` and `writeOfflineEvaluationResult`. `Protocol_accessibility_wcag2ict` extends these methods to report success criterion violations.

The code as discussed in this section differs from the typical protocol in that it delegates the test work to an `Evaluator`. Apart from this design decision, the accessibility protocols do not differ significantly from other protocols in TESTAR. In particular, the action selection algorithm can be any algorithm that TESTAR supports. We used random action selection as the default and did not investigate other algorithms because that would be premature optimization.

8.3. ACCESSIBILITY STANDARDS

Evaluating software on its accessibility using TESTAR and with WCAG2ICT as the standard requires an implementation of the `Evaluator` interface (see Section 8.2). This is where the oracles based on success criteria are translated into code. In other words, this is the implementation of the accessibility standard we selected in Chapter 6. Before discussing our implementation, we illustrate it with the code for two oracle oracles, one on-the-fly and one off-line.

Listing 8.1: On-the-fly oracle procedure for success criterion 3.1.1: Language of Page

```
public EvaluationResults evaluate(List<Widget> widgets) {
    EvaluationResults results = new EvaluationResults();
    SuccessCriterion sc = getSuccessCriterionByName("Language of Page");
    for (Widget w : widgets) {
        if (AccessibilityUtil.isWindow(w) {
                && AccessibilityUtil.getLanguage(w) == 0)
            results.add(new WCAG2EvaluationResult(
                    sc, WCAG2EvaluationResult.Type.ERROR,
                    "Missing top−level language identifier", w));
        }
        else {
            results.add(evaluationPassed(sc));
        }
    }
    return results;
}
```

Listing 8.2: Off-line oracle procedure for success criterion 2.4.2: Page Titled

```
private static final int SAME_TITLE_THRESHOLD = 3;

public EvaluationResults query(GraphDB graphDB) {
    EvaluationResults results = new EvaluationResults();
    SuccessCriterion sc = getSuccessCriterionByName("Page Titled");
    String gremlinTitleCount = "_().has('@class','Widget')" +
            ".has('" + WCAG2Tags.WCAG2IsWindow.name() +"',true)" +
            ".groupCount{it." + Tags.Title.name() + "}.cap";
    List<Object> titleCounts = graphDB.getObjectsFromGremlinPipe(
            gremlinTitleCount, GremlinStart.VERTICES);
    // the list contains one map with title counts
    Map<String, Long> titleCount = (Map<String, Long>)titleCounts.get(0);
    boolean hasViolations = false;
    for (Entry<String, Long> entry : titleCount.entrySet()) {
        if (entry.getValue() > SAME_TITLE_THRESHOLD) {
            hasViolations = true;
            results.add(new WCAG2EvaluationResult(
                    sc, WCAG2EvaluationResult.Type.WARNING,
```

```
                    "Possible ambiguous title \"" + entry.getKey() +
                    "\" appeared " + entry.getValue() + " times"));
            }
        }
        if (!hasViolations) {
            results.add(evaluationPassed(sc));
        }
        return results;
    }
```

Listing 8.1 shows the implementation of an on-the-fly oracle in method `evaluate`. The oracle is for success criterion 3.1.1: Language of Page (see Section 7.4.4). After declaring some variables, the method starts walking through the given widgets. If a widget is a window and has a language identifier of 0, then the method adds an error to the evaluation results. There can be multiple such results if a window has sub-windows, for example dialogue boxes. When the widget is not a window or the window has a non-zero language identifier, the result is a pass. There can thus be as many passes as there are windows. The method ends by returning the evaluation results. The other on-the-fly oracles are implemented in the same way, although the types and number of properties they test differ.

Listing 8.2 shows the implementation of an off-line oracle in method `query`. The oracle is for success criterion 2.4.2: Page Titled (see Section 7.4.3). The method starts in the same way as the one we just discussed, but defines a Gremlin query in a string. The query starts by selecting all vertices that are widgets and that have the "WCAG2IsWindow" property set. These widgets are windows. The widgets are grouped by their "Title" property. The result is a map of titles and the number of times they appear in the graph database. The method loops through these counts. Any count above a predefined threshold is a warning. If all titles have been tested and there were no warnings, the result is a pass. The pass means that the query yielded no violations, so there is a single pass for the whole method instead of a pass for every title count that is below or equal to the threshold. Again, the method ends by returning the evaluation results. The only other implementation of an off-line oracle has a similar structure, but its Gremlin query is longer.

Figure 8.4 displays the WCAG 2.0 classes. To limit the size, we included only one concrete guideline. In the code, all guidelines have a corresponding class. `WCAG2ICT` is the umbrella class, `AbstractPrinciple` and `AbstractGuideline` abstract principles and guidelines, respectively. Instances of `SuccessCriterion` represent concrete success criteria.

`AbstractPrinciple`, `AbstractGuideline` and `SuccessCriterion` share a set of common attributes. They all have a name and a number from WCAG 2.0. Guidelines and success criteria further have a parent item. Principles form the root of their respective trees. `ItemBase` encapsulates these details. Each WCAG 2.0 item inherits from this abstract class. `WCAG2ICT` itself is not considered to be a WCAG 2.0 item.

Concrete principles inherit from `AbstractPrinciple` and concrete guidelines inherit from `AbstractGuideline`. A `SuccessCriterion` is instantiated with the appropriate information by the concrete guideline that contains it. For example, `TextAlternativesGuideline` instantiates a `SuccessCriterion` with the name "Non-text Content". This design decision is because of efficiency considerations. On-the-fly evaluation as well as deriving actions typically requires a full walk through the relevant widgets in a `State`. A guideline evaluates

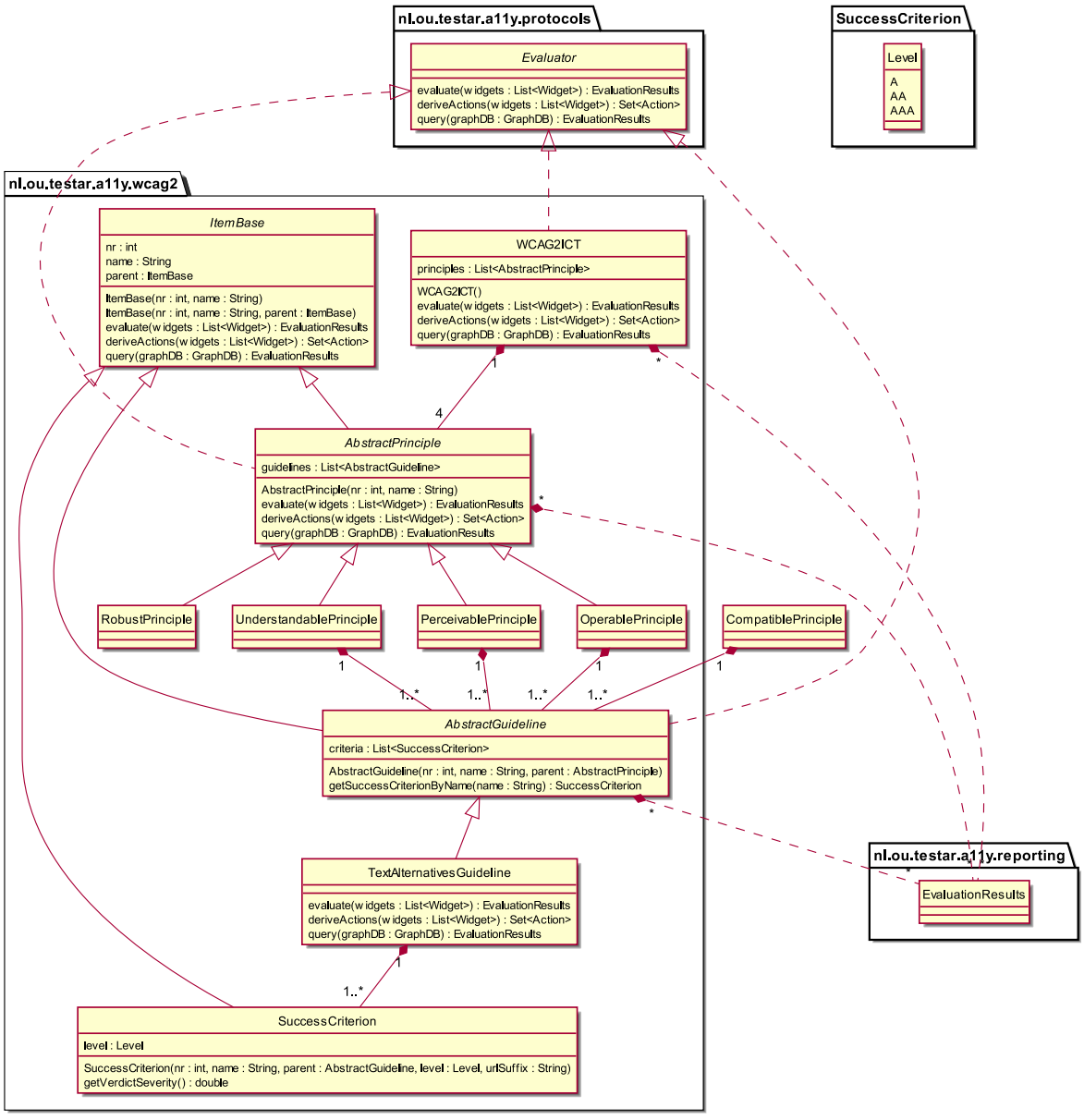


Figure 8.4: Class diagram showing the WCAG 2.0 accessibility standard classes

all of its success criteria in one walk and derives actions in another walk. Off-line evaluation does not have an efficiency impact, but a guideline still performs this task to remain consistent.

`AbstractGuideline` instances contain the oracle procedures and information for on-the-fly and off-line evaluation and the rules to derive actions. `SuccessCriterion` is only used when reporting results (see Section 8.4. A consequence of this design decision is that `SuccessCriterion` does not implement `Evaluator`. Therefore, it is not possible to evaluate individual success criteria. The lowest granularity is at the guideline level, followed by principle and WCAG2ICT as a whole.

`ItemBase`, the base class for WCAG 2.0 items, provides default implementations for all three `Evaluator` methods. The default behaviour is to do nothing and return an empty collection of evaluation results or set of actions, depending on the method. `WCAG2ICT`, `AbstractPrinciple` and `AbstractGuideline` have recursive implementations of these methods, as Figure 8.2 showed:

1. `AbstractGuideline` implements `evaluate` and `query` to return empty `EvaluationResults` and implements `deriveActions` to return an empty set of actions. Sub-classes should provide their own implementations. This is the stopping point of the recursion.

2. `AbstractPrinciple` calls these methods on all guidelines under it and returns the accumulated results.

3. `WCAG2ICT` calls these methods on all principles under it and returns the accumulated results.

A `SuccessCriterion` can return its severity based on its conformance level through the method `getVerdictSeverity`. This method provides compatibility with the traditional TESTAR `Verdict` (see Section 2.3). The return value falls within the range of `Verdict` severities, 0 to 1 inclusive. Conformance level A has the maximum severity, 1, and the severities for level AA and AAA are each an equal amount lower. Because the maximum severity is 1 and there are three conformance levels, the severity is lowered by $\frac{1}{3}$ per level. So level AA is $\frac{2}{3}$ and level AAA is $\frac{1}{3}$. If we number level A, AA and AAA sequentially starting at 0, the return value is:

$$severity = MAX\_SEVERITY - level * \frac{1}{3}$$

`AbstractGuideline` has a helper method, `getSuccessCriterionByName`, to easily find a success criterion given its name. The success criterion must belong to the guideline the method is called on. When returning an evaluation result, this is the method that is used to find the evaluated success criterion. For evaluations performed with the WCAG 2.0 implementation, it is always possible to get the evaluated `SuccessCriterion`.

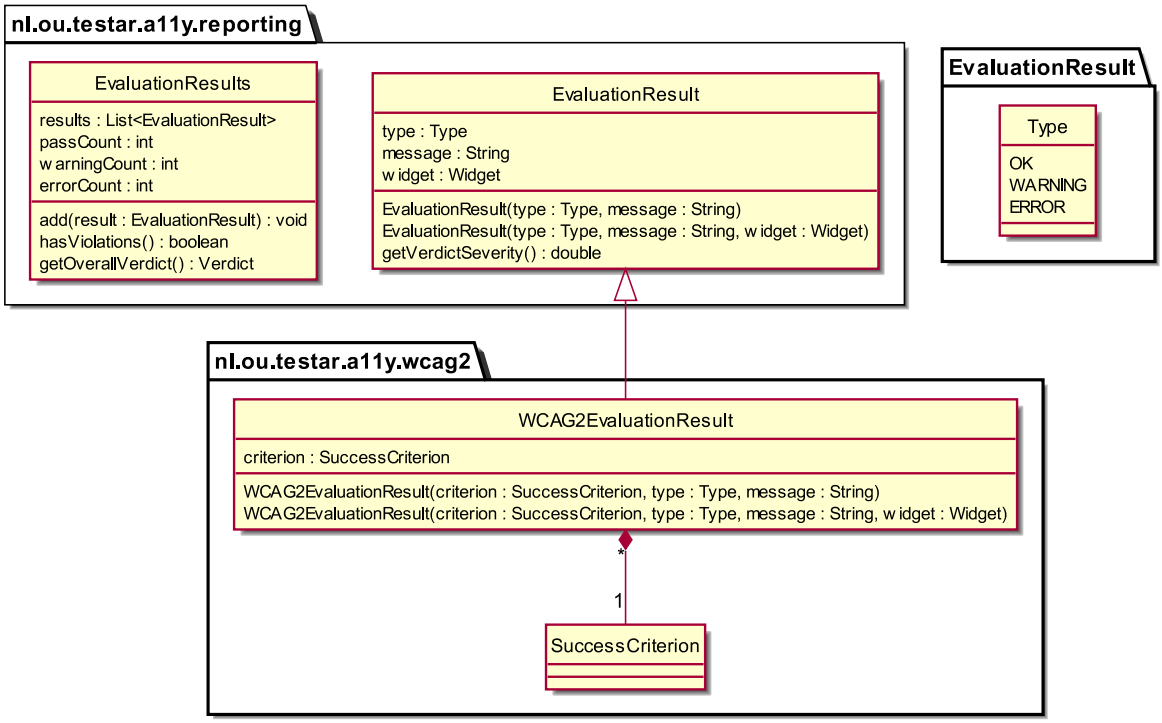


Figure 8.5: Class diagram showing the evaluation results classes

8.4. EVALUATION RESULTS

Figure 8.5 shows the evaluation results classes. Test protocols in TESTAR rely on the `Verdict` class to signal problems and designate the problem severity (see Section 2.3). We added three classes to store more descriptive information about the evaluated accessibility rules.

`EvaluationResult` stores additional information about a result. The type indicates the outcome of the evaluation (error, warning or pass, see Section 2.4). A text description helps experts to understand the result. Optionally, if the evaluation was performed with a specific widget, that information is also stored. `WCAG2EvaluationResult` further stores the evaluated `SuccessCriterion`. `EvaluationResults` collects `EvaluationResult` classes and also maintains pass, warning and error counts. The `evaluate` and `query` methods from the `Evaluator` interface return `EvaluationResults`.

Analogous to `SuccessCriterion`, `EvaluationResult` has a method `getVerdictSeverity`. It serves the same purpose, namely to provide compatibility with the `Verdict` class in TESTAR. The default implementation returns the minimum severity, 0, for passes and the maximum severity, 1, for violations. The `WCAG2EvaluationResult` implementation instead returns the value from the associated `SuccessCriterion`. `EvaluationResults` has a method `getOverallVerdict` that computes an overall `Verdict` for all collected results. It does this by finding the `EvaluationResult` with the highest severity and constructing a `Verdict` with that severity and a generic message.

An `EvaluationResult` can be added to `EvaluationResults` by calling `add`. Based

Figure 8.6: Sequence diagram demonstrating the execution flow to evaluate a WCAG 2.0 guideline

on the added results, if any, `hasViolations` returns if there are results that are violations (error or warning). It is also possible to access the collected `EvaluationResult` classes and to get the exact count of passes, errors or warnings. `EvaluationResults` facilitates the communication between TESTAR and the expert.

To illustrate how the components discussed so far fit in the design, Figure 8.6 shows the execution flow to perform on-the-fly evaluation with one WCAG 2.0 guideline. The processes to derive actions or to perform off-line evaluation are similar, so the figure does not show them. In TESTAR, `WCAG2Protocol` is actually called `Protocol_accessibility_wcag2ict`, but for readability we use the shorter name.

8.5. UTILITIES

Figure 8.7 displays the two utility classes. `AccessibilityUtil` supports accessibility evaluation. It contains platform-specific accessibility utilities in the form of attributes and methods to interface with the accessibility API, hence the dependency on the platform. Test protocols and accessibility standard classes use these accessibility utilities. `HTMLReporter` supports writing the evaluation report. It is a simple HTML report writer. Only test protocols currently use this class.

`AccessibilityUtil` contains static attributes and methods. The figure lists the most significant ones. The accessibility utility class is in the package `nl.ou.testar.a11y.windows`. Since we only have an implementation for Windows using UIA, we did not abstract the

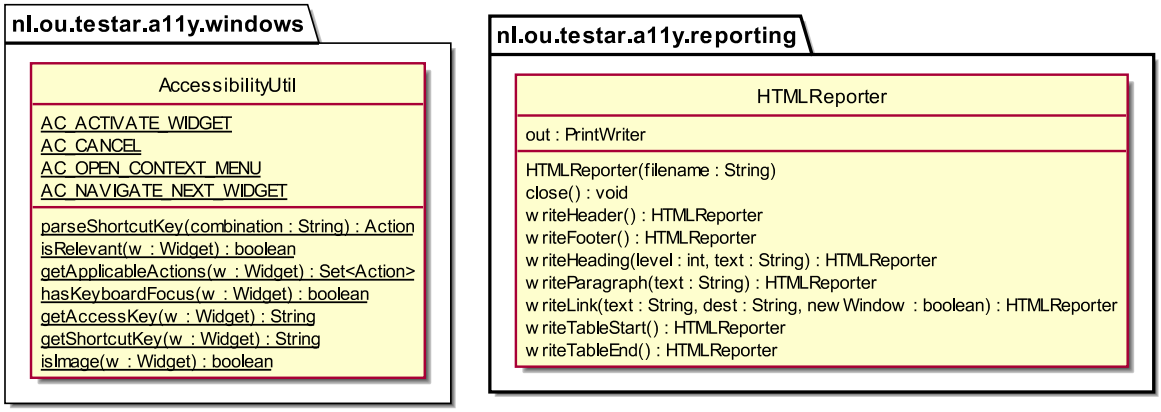


Figure 8.7: Class diagram showing the utility classes

utilities into an interface with implementations for different APIs. We did not investigate if there is a common set of operations in all APIs that TESTAR supports. To support a new API, the `AccessibilityUtil` methods must once be specified in an interface and the callers of these methods must be adapted to use the interface. These methods have to be implemented for each new API. `AccessibilityUtil` is the only platform-specific code in our implementation.

There are several accessibility utilities. A number of keyboard actions have been predefined, for example to activate a widget, cancel an operation, open the context menu or navigate through widgets. To parse arbitrary strings containing shortcut key descriptions into actions, `parseShortcutKey` can be used. It takes a single key or a key combination, parses the individual key names into key codes and combines them into a compound `Action` that TESTAR can execute.

The remaining accessibility utility methods work on widgets. The evaluation process uses only relevant widgets (see Section 8.2). The default way to decide if a widget is relevant is through the `isRelevant` method.

When deriving actions, `getApplicableActions` takes a widget and looks up the possible actions for it. `AccessibilityUtil` has definitions of standard actions for roles that occur frequently, such as editable widgets which support the arrow keys. The returned set of applicable actions is not complete, as it only looks at roles. Shortcut keys and other properties have their own methods.

The last set of methods return widget properties. They can directly return the value from the API, as with `hasKeyboardFocus`, or base their return value on the widget role, such as `isImage`. `getShortcutKey` returns the accelerator key, if any, but it also tries to gather a shortcut key from the widget name if the accelerator key is unset. This situation appears to occur often with menu items. `getAccessKey` returns the access key.

TESTAR generates the evaluation report as an HTML file using a simple `HTMLReporter` class. The `HTMLReporter` creates a new output file upon construction. `close` closes the file. The other methods all write specific HTML elements. Some elements have no parameters

and are simply written to the report, such as the HTML file header and footer or the start and end tags of elements like tables. Other elements can contain text between their opening and closing tags, such as paragraphs. Elements can also have extra parameters that determine the tag that is written, such as with headings, or that become attributes on the tag, such as with links. The methods that write to the output file return the `HTMLReporter` instance itself, thereby allowing method call chaining.

8.6. CONCLUSION

The implementation of accessibility evaluation in TESTAR consists of three main extensions and two utility classes. The test protocols coordinate the test process. Accessibility standard classes perform the evaluation through on-the-fly and off-line oracles and derive actions. Evaluation results classes store information to report to the expert. The utility classes provide supporting functionality.

Our implementation allows complete access to the information available from the accessibility API. As noted for UIA in Chapter 7, this makes it possible to test some accessibility rules, but there are also limitations. The information comes either from the state or from the graph database. Generally speaking, not all required properties are always available from the accessibility API. Examples of what UIA lacks are widget foreground and background colour properties. This makes it impossible to evaluate the contrast level, among potential other rules. Extending the implementation beyond the use of accessibility APIs, for example by capturing and analysing screenshots, could provide ways to evaluate more rules.

9

CASE STUDY

In this chapter we present the results of a case study where we apply the ideas presented in the previous chapters on a real-world application. In Section 9.2, we compare the information available within the graph database with the output originally provided by TESTAR. This addresses research question Q3. In Section 9.3, we analyse the performance of the graph database. This addresses research question Q4. In Section 9.4, we validate the accessibility report TESTAR generates on its usefulness to experts. This addresses research question Q9.

9.1. SUT SELECTION

In our case study, we apply TESTAR on VLC[1]. VLC is an application to play audio and video content with a very rich user interface. The rich interface makes VLC suitable to become the SUT. The large number of different user interface components will provide a realistic load on the database. It is also a good application to test drive the accessibility evaluation. In our research we used version 2.2.6.

[1] https://www.videolan.org

9.1.1. THE EXECUTION ENVIRONMENT

In order to provide a controlled environment, the SUT is installed on an instance of VirtualBox (Section 9.2 and 9.3) or VMware Player (Section 9.4). Table 9.1 and 9.2 present the properties of the VirtualBox and VMware Player instances, respectively.

Property	Value
Operating system	Window 10 64-bit
Number of CPUs	2
Memory	4 gigabyte
Disk size	32 gigabyte[2]

Table 9.1: Properties of the VirtualBox instance used for testing

Property	Value
Operating system	Window 10 64-bit
Number of CPUs	2
Memory	3 gigabyte
Disksize	127 gigabyte (max.)

Table 9.2: Properties of the VMware Player instance used for testing

[2]The data of TESTAR will be stored on a shared drive for the performance analyses

9.2. EVALUATING THE EFFECT OF STORING INFORMATION IN A GRAPH DATABASE

To demonstrate the benefits of storing the results in a graph database compared to the current way of storing results (text, visual graph), we first discuss the current output of TESTAR. Table 9.3 presents an overview of the types of outputs currently provided by TESTAR.

Output artefact	Description
Graph	a .dot file which can be used with Graphviz[3] to generate visualization of the execution graph
Sequence	Binary representation of state information stored in TESTAR and used to replay.
Logs	Textual information about the actions performed during the execution of TESTAR
Metrics	Indicators about the test performance. (see Section 5.6 of the TESTAR manual [37], for more information)

Table 9.3: Output artefacts of TESTAR

Currently, the graph and logs contain information that can be used to analyse data outside of TESTAR. The folders containing sequence data are used for replay. In the remainder of this section, we will demonstrate how we can query the graph database using the Gremlin console provided by OrientDB. We aim to obtain the same information as provided by TESTAR. Having all test output data in the graph database allows us to write powerful queries. However, the data in the database is also redundant to the existing output of TESTAR. By trying to obtaining all the existing output data through queries we could verify if the graph database could replace the existing output.

Besides the answer to the question if it is possible to retrieve the data, we will also investigate the usability of Gremlin. We will look how easy it is to create a query, understand the feedback, translate our ideas into a query and present the results in a readable format.

[3] http://www.graphviz.org

9.2.1. USING THE GRAPH DATABASE

To answer research question Q3, we will try to retrieve the information presented in the current output by applying traversals on the graph database. We start by looking at the information stored in the folder "Logs". For each sequence, TESTAR stores 5 types of output files. Table 9.4 lists these types.

Log types	Description
sequence<number>	Log file containing detailed information about the actions executed in the sequence
sequence<number>_clusters	presents information about how actions and states are clustered
sequence<number>_curve	presents statistics per action
sequence<number>_stats	presents overall statistics for the sequence
sequence<number>_testable	presents detailed information about the executed actions

Table 9.4: Various output formats in the folder "output\logs"

We will try to generate the same output using the Gremlin console provided with OrientDB. Our goal is to retrieve the information as a JavaScript Object Notation (JSON) object. JSON is selected since it is a standardized format which makes it easy to present data on, among others, websites. It is well structured and easy to read or parse.

COMPARISON WITH TEST OUTPUT FROM LOGS

Property	Description
@class	type of the vertex
Resources	contains the resources used during the execution of the action.
Representation	detailed information of the executed action which is aggregated by. TESTAR
sequenceNumber	Number that determines the position in the order of the task execution

Table 9.5: Properties used in the queries presented in this section

The information, found in the artefact "sequence<number>" (see Table 9.4), can easily be retrieved by listing the property "Representation" for all actions. However, this property was not available in the set of tags which are stored with an Action. To resolve this issue, we applied the technique represented in Section 5.1. Listing 9.1 shows the code that can be used within the Gremlin console to retrieve the required information as a JSON object. A brief description of Gremlin is provided in Section 3.4.2.

Listing 9.1: Gremlin query to retrieve information about the executed actions

```
list = G.V.has("@class","Action")
   .map("@class","Resources","Representation","sequenceNumber")
   .toList()
json = groovy.json.JsonOutput.toJson(list)
```

The query presented in Listing 9.1 starts with the step G.V which finds the traversers for all vertices. This set is reduced by the step *has*("*@class*","*Action*") to a set of traversers for the vertices of the type "Action". The next step "maps" a selection of the properties for each traverser to a map presentation. The result is collected in a list. In the next statement, this list is transformed into a JSON object. Listing 9.2 presents a part of the result of the query.

Listing 9.2: Piece of the result from the query presented in 9.1

```
[
 {
  "sequenceNumber": 0,
  "@class": "Action",
  "Representation": "LC (  WC1szr01y3a1643044285, UIACheckBox,
  Regularly check for VLC updates ) [ ]",
  "Resources": "{ userCPU:62,systemCPU: 187,ramBase:28016.0,ramPeak:0.0}"
 },...
]
```

Listing 9.3 shows the output for the same action which was recorded in the original output from TESTAR. The big difference is the representation part. In Listing 9.3, the detailed representation is presented while in Listing 9.2 shows the condensed output for the representation. Also the CPU and memory information is not presented in the same way. This is caused by an implementation failure of the Resource tag in the class `AbstractProtocol`.

Listing 9.3: Output for the same action retrieved from "sequence01.log"

```
Executed [1]: action = AC1b7g5l3802595899041 (AA1b7g5l3802595899041) @state = SC1qlmffo4213155223051 (SR1nui0x83fa1331420749)
    SUT_KB = 28492, SUT_ms = 16 x 16 x 1.23%
    ROLE = LeftClickAt
    TARGET =
        WIDGET = WC1szr01y3a1643044285, WR88rmk0b1934231702, WTza4if52a723259854, WPlgn37c361924066633
        ROLE = UIACheckBox
        TITLE = Regularly check for VLC updates
        SHAPE = Rect [x:509.0 y:586.0 w:315.0 h:17.0]
        CHILDREN = 0
        PATH = [0, 0, 2, 1]
    DESCRIPTION = Left Click at 'Regularly check for VLC updates'
    TEXT = Compound Action =
        Move mouse to (666.5, 594.5).
        Press Mouse Button BUTTON1
        Release Mouse Button BUTTON1
```

The information, found in the artefact "sequence_clusters<number>" (see Table 9.4), can be retrieved using queries 9.4 and 9.5.

Listing 9.4: Gremlin query to retrieve State clusters

```
list =G.V.has("@class","AbsState").in.groupBy{it.Abs_R_ID}{it.ConcreteID.next()}.cap.next()
json = groovy.json.JsonOutput.toJson(list)
```

Listing 9.5: Gremlin query to retrieve Action clusters

```
G.V.has("@class","AbstractAction").in.groupBy{it.AbstractID}{it.ConcreteID.next()}.cap.next()
json = groovy.json.JsonOutput.toJson(list)
```

Both the queries presented in Listing 9.4 and Listing 9.5 follow the same pattern. The first step provides a traverser for each vertex. In the next step, the set of traversers is reduced by applying the step *has*("*@class*","*AbstractState*") or *has*("*@class*","*AbstractAction*"). These steps result in a set of traversers for each AbstractState or AbstractAction. For each

traverser the incoming Edges are grouped by the abstract id resulting in list of abstract Id's combined with the concrete Id's of the elements that share the concreteID.

Listing 9.6: Snippet of the result from the query presented in 9.4

```
{
...
"SR15vmxu0b213146233953": [
 "SCxhxbkzb7d2917862295",
 "SCa464q9b7b4077626259",
 "SC99b36ub7c3703555025"
],
...
}
```

Listing 9.7: Snippet from "sequence99_clusters.log"

```
SR15vmxu0b213146233953 contains:
  (1) SCa464q9b7b4077626258 (2) SCxhxbkzb7d2917862294 (3) SC99b36ub7c3703555024
```

Listing 9.6 shows a snippet for the result of the query presented in Listing 9.4. Listing 9.7 shows the output for the same abstract state. This latter information is retrieved from the sequence<number>_clusters.log file which is part of the original output of TESTAR. Looking at the listings it is clear that both show the same information. Listing 9.7 adds order information. Currently, this information is not available in the graph database.

The information, found in the artefact "sequence_curve<number>" (see Table 9.4), cannot be retrieved from the graph database using the data model which is presented in Section 4.2. The artefact contains statistic information which is collected at the end of each `Action` execution. A solution would be to model this data as a set of properties on the "execute" Edge. Another solution would be to model the data in the `TESTAREnvironment` into the graph database. This exercise is left as future work.

The information, found in the artefact "sequence_stats<number>" (see Table 9.4), can partially be retrieved through queries on the model in the graph database. The number of unique states, unique action, abstract states and abstract actions can simply be found with a query that counts the vertices of these types. For example, the Query, presented in Equation 9.1, counts the unique actions. However, the artefact "sequence_stats<number>" also contains a number of items which cannot be retrieved from data model which is implemented during our research. Items like unexplored actions and unexplored states are calculated within the `TESTAREnvironment` which is not exposed in the graph database.

$$G.V.has(\text{“}@class''\text{,“}Action''\text{).}count() \tag{9.1}$$

The information, found in the artefact "sequence_testable<number>" (see Table 9.4), could be retrieved from the graph database since all information is available starting from an `Action` vertex. Constructing the proper query is left to the reader.

USING GREMLIN QUERIES WITH ORIENTDB

During the writing of the Gremlin queries, we learned that the language has a high learning curve. Upfront, it seems quite convenient that you can simply concatenate different steps which will in the end provide the required result. However, we found the following problems;

1. The Gremlin version, which is delivered with OrientDB 2.2.x, is not the latest released by Tinkerpop[4]. Therefore, it is not possible to leverage all the features available. It was also harder to apply the information provided by research papers like [33].
2. The Gremlin console, provided by OrientDB, provides poor feedback when a query fails.
3. When the proper documentation was found, it showed that various steps resulted in slightly different result types. The returned types where poorly documented.

OrientDB also provides an SQL dialect with features which allows the user to traverse a graph[5]. In the upcoming release of OrientDB (version 3.x). support for the official Tinkerpop version of Gremlin is provided. In a future project, the OrientDB version should be upgraded and the capabilities of the new Gremlin version need to be investigated.

[4] http://tinkerpop.apache.org
[5] http://orientdb.com/docs/2.2.x/SQL.html

COMPARING THE VISUALIZATION OF ORIENTDB WITH THE VISUALIZATION WITHIN TESTAR

Besides the information contained in the log folder, TESTAR produces more outputs, as summarized in Table 9.3. The graphs stored in the graphs folder are constructed from the internal model contained in `GraphState` by TESTAR and stored in the ".dot" format[6].

This format is quite straightforward. It lists the nodes and edges of a graph and the relation between them. For the graph that contains screenshots, the screenshot property from the `State` is used. All[7] information is available within the data model of the graph database.

OrientDB also provides its own visualization of a graph. To use this, the database must be imported into an OrientDB database that runs on a server and can be accessed using the web interface (see Appendix C for information on export/import). Figure 9.1 shows an example showing only 2 states, there widgets and a selection of the actions. The visualization of OrientDB is only of use for a limited number of objects. When the number increases the web page with the results becomes very slow and the information becomes to much cluttered.

As a solution it should be investigated if it is possible to present the data at an higher abstraction level. For instance, only showing the states and there relation. This subject is left for future research.

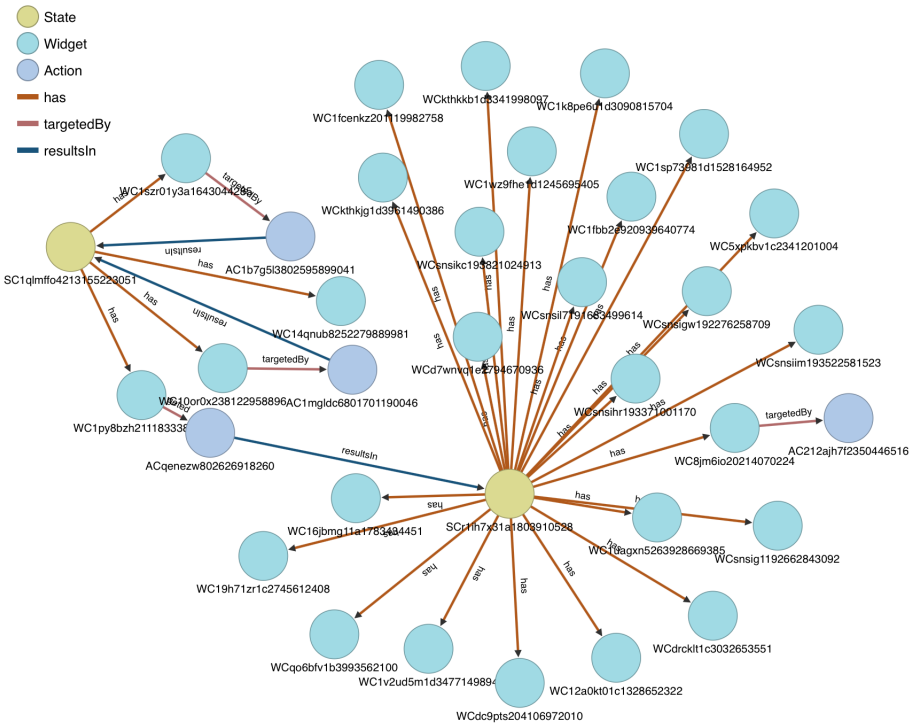


Figure 9.1: Example showing the visualization of 2 States, their related Widgets and a selection of Actions

[6] https://graphviz.gitlab.io/_pages/doc/info/lang.html
[7] For screenshots, the path to the actual image is stored within the model

REPLAYING TEST SEQUENCES

The information stored in the binary sequence files contains a sequence of "fragments". These fragments are key-value pairs containing information about different properties found during the execution of a test. TESTAR uses this information when running in replay mode.

Part of the information contained in these fragments is also contained in the data model for the graph database. However, the graph database does not contain all the required information. Since the fragments are dedicated to the replay-mode, which is internal to TESTAR, it does not benefit the user when the information modelled in the fragments would be stored in the graph database.

9.2.2. SUMMARY ON USING THE GRAPH DATABASE

In this section we showed how the graph database can be used to retrieve information that is already logged by TESTAR. To answer question Q3, we can conclude that the graph database solution can potentially replace the current output of TESTAR. Using our prototype implementation, most information can be retrieved. Some elements are not available in the data model implemented for the graph database. This missing information can be made available by storing additional information in the database. For instance the properties stored within the internal model of TESTAR.

Besides the possibility to retrieve existing information, the database can provide opportunities to retrieve other information about the SUT in a off-line analysis. Therefore, the existing output and the graph database can coexist providing complementary capabilities to TESTAR.

9.3. PERFORMANCE IMPACT FOR THE USE OF A GRAPH DATABASE

In order to answer question Q4, we measure the performance of the database. For the thesis, performance is divided into two categories;

1. The time it takes to store data into the database
2. The amount of disk space required

The timing is measured in two ways. First, we perform a micro benchmark using Java Micro benchmark Harness (JMH)[8]. JMH is a benchmarking tool kit provided by the openjdk community. We use JMH to measure the execution time of the API defined in the module graphdb. We explain the set-up of this test in Section 9.3.2 and the results in Section 9.3.3.

Second, we perform an application benchmark. This benchmark is measured running TESTAR in a series of scenarios. For each action executed by TESTAR, the duration of the database operations is measured. This measurement will provide information on the execution time of each single database operation as well as it will also provide information on how the execution time of these operations changes when the dataset becomes larger. We present the results of this experiment in Section 9.3.4.

Section 9.3.5 presents the results on disk space usage. Both disk space usage and real-world timing are measured using the same series of TESTAR runs. In each run an increasing number of execution steps is performed. This way we gather information which is sufficient to predict the timing and disk space requirements over a larger period of time. We present the conclusions of the experiments in Section 9.3.6.

In this section we look a the time it takes to store "artefacts". With artefacts we mean a `State`, `Action` or `Widget` unless stated otherwise.

[8] `openjdk.java.net/projects/code-tools/jmh`

9.3.1. SET-UP

All tests are executed on an instance of Windows 10 running in VirtualBox[9]. Table 9.6 shows the configuration of the PC.

Component	Specification
CPU	Intel Core I7-3770 3. gigahertz
Memory	8 gigabyte
Operating System	Window 10 Education 64 bits
Hard disk	500 gigabyte SSD

Table 9.6: Hardware configuration of the test system

In order to improve the reproducibility of the tests and not to harm, for instance destroy important files, the host system, the tests are executed on a Windows 10 instance running within VirtualBox. Table 9.1 shows configuration which applies to this virtual machine.

9.3.2. THE BENCHMARKS

The performance of of the graph database when used within TESTAR, is measured using two indicators;

1. The execution time of a database operation.
2. The disk usage of the database.

The benchmark test used to measure these indicators are introduced in this section.

THE SYNTHETIC BENCHMARK

The JMH benchmark will be executed from the development environment. The build configuration for the module `graphdb` is extended with support to run the benchmarks. The benchmarks themselves are implemented in the sources found in the "graphdb/jmh/java/nl.ou.testar" folder. They can be executed by running the command `./gradlew jmh`. JMH has a number of configuration options. These options are explained at [27]. The used configuration is presented in Table 9.7.

[9] https://www.virtualbox.org

Setting	Description	Value
Warmup	Number of iterations executed before the measurement starts. This is needed two minimize the impact of the JVM	30
Fork	The number of times the test is executed. Each time the the test is started from scratch.	10
Measurement iterations	The number of times the test is executed within one fork	10
Measurement time	The time an iteration is allowed to last	1
Measurement timeUnit	The time unit in which the execution of the test is measured	TimeUnit.Milliseconds
BenchmarkMode	Type of benchmark which is executed. SampleTime, Throughput, AverageTime and SingleShot	SampleTime
OutputTimeUnit	Time unit to present the result	TimeUnit.Milliseconds

Table 9.7: Configuration of the benchmarks

The executed benchmarks all focus on the API of the interface `GraphDBRepository` (see Appendix D). The goal is to measure the average execution time of those API calls. Table 9.8 provides an overview of the defined benchmarks. Note, only a selection (those APIs which are executed most often) are measured in the benchmark.

Benchmark	Description
AddSateBenchmark.testAddState	How long does it take to store a `State`
AddSateBenchmark.testAddStateTwice	How long does it to update a `State`
AddActionBenchmark.testAddSingleAction	How long does it take to store an`Action` in the database that is applied on a `Widget`
AddActionOnStateBenchmark.testAddSingleAction	How long does it take to store an `Action` in the database this is applied on a `State`
AddWidgetBenchmark.testAddSingleWidgetToState	How long does it take to store an `Widget` in the database
Linearitybenchmark.testAddStateLiniairity	Investigate the Impact of adding new states to an existing database already containing states.

Table 9.8: Description of the benchmarks

The results of the synthetic benchmark are presented in Section 9.3.3.

THE REAL-WORLD BENCHMARK

For the real-world benchmark, TESTAR is used to test VLC[10]. VLC is introduced in Section 9.1. TESTAR is evaluated in several scenario's, presented in Table 9.9. For each scenario, the results are stored in a graph database located on the local file system. This will result in a folder for each scenario. The size of the folder is examined. This is a measure for the amount of disk space required to run the scenario. The results for all scenario's are combined in a graph. It is expected that the results can be extrapolated to predict the results for other sizes of the scenario.

In order to investigate the time required to store the information in the database, each action on `GraphDBRepository` is logged including the amount of time required to perform the action. The results of the real-world benchmark are presented in Section 9.3.4. Section 9.3.5 presents a discussion on the required disk space for the database.

Number of Sequences	Number of Actions	Scenario
1	100	desktop_generic_graphdb
10	100	desktop_generic_graphdb
50	100	desktop_generic_graphdb
100	100	desktop_generic_graphdb

Table 9.9: Test scenarios which will be executed with TESTAR. The settings for this scenario can be found in Appendix F.

[10] https://www.videolan.org

9.3.3. RESULTS FOR THE SYNTHETIC BENCHMARK

Table 9.10 shows the results of the benchmarks for the virtual machine presented in Table 9.1 running on hardware presented in Table 9.6. For each test, the average execution time and the variance (the error) is listed.

Benchmark	Iterations	Time (ms)	Error (ms)
AddActionBenchmark.addSingleAction	N/A	8.367	± 1.691
AddSateBenchmark.testAddState	N/A	6.138	± 1.053
AddSateBenchmark.testAddStateTwice	N/A	9.331	± 1.347
AddWidgetBenchmark.addSingleWidgetToState	N/A	12.064	± 2.102
Linearitybenchmark.testAddStateLiniairity	1	0.001	± 0.001
Linearitybenchmark.testAddStateLiniairity	2	5.580	± 0.982
Linearitybenchmark.testAddStateLiniairity	4	10.088	± 1.719
Linearitybenchmark.testAddStateLiniairity	8	18.634	± 4.168
Linearitybenchmark.testAddStateLiniairity	16	20.883	± 3.544
Linearitybenchmark.testAddStateLiniairity	32	29.493	± 3.996
Linearitybenchmark.testAddStateLiniairity	64	68.794	± 4.890
Linearitybenchmark.testAddStateLiniairity	128	144.181	± 8.622
Linearitybenchmark.testAddStateLiniairity	256	371.196	± 11.297
Linearitybenchmark.testAddStateLiniairity	512	1131.214	± 29.199
Linearitybenchmark.testAddStateLiniairity	1024	4124.888	± 126.170

Table 9.10: Result of the benchmark performed with JMH. Each benchmark presents the average results over 100 measurements. The SampleTime benchmark was used.

REFLECTION ON THE SYNTHETIC BENCHMARK

The simplest operation of the module `graphdb` is `addState`. It is expected that this operation takes the least amount of time. The results, presented in Table 9.10, confirm this assumption. Adding a state twice does not require double the amount of time since the second add operation only requires a lookup. The operation `addSingleAction` takes longer. This operation requires to additional lookups for a vertex compared to the operation `addState`. The operation `addSingleWidgetToState` takes longer as the operation `addSingleAction`. Although the operation `addSingleWidgetToState` requires less lookups, the operation adds an edge to three abstract vertices. an Action only has one abstract type. The linearity-benchmark shows to the required to store multiple states. Figure 9.2 shows the results in a graph form. We plotted a trend line through the points. Equation 9.2 presents the formula for the trend line. The formula has a good fit ($R = 0.9999$) and shows that storing a state has a polynomial behaviour.

$$y = 0.0035x2 + 0.4295x + 15.922 \tag{9.2}$$

The error presented in Table 9.10 is large. In order to pinpoint the cause of the error, we repeated the test with an in memory database. Although the average execution time will be

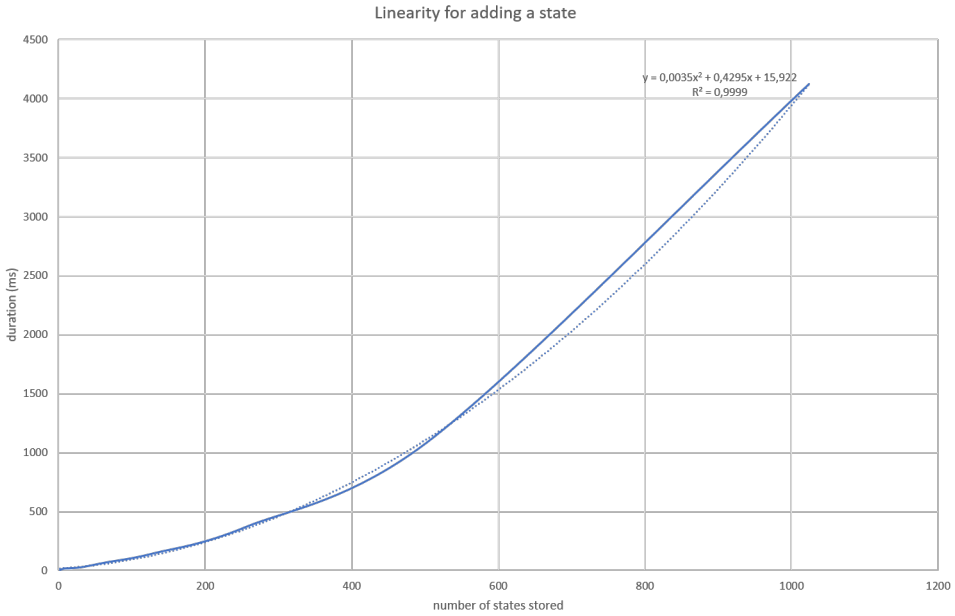


Figure 9.2: Linearity for adding a state

lower, as expected, the error in these measurements is even higher. Most likely cause is the fact that OrientDB is not very deterministic in its execution time.

9.3.4. RESULTS FOR THE REAL-WORLD BENCHMARK

EXECUTION TIME IN A REAL-WORLD BENCHMARK

In order to get execution time information for calls to the API of `GraphDBRepository`, we added log statements at the end of each invocation. The log statement records the time it took to perform the interaction with the graph database. As input for our analyses we used the output of the TESTAR run where we executed 100 sequences each consisting of 100 executed actions. During the execution, the data was logged to a text file, the text file was filtered to split the log into separate files for actions, states and widgets and we performed a statistical analysis with Excel. The results of this analysis are presented in this section.

Storing an `Action` Figure 9.4 and Figure 9.3 (detail) show the execution time for each stored action during the real-world benchmark with 10,000 samples. When we compare these measurements with the results of the synthetic benchmark, the duration of the real-world benchmark is around the same value at the start of the test. However the variation of the different samples is a lot bigger (over 20 milliseconds). This could be caused by the SUT which is running on the same machine. It is expected that the execution time will increase as more data is stored in the database. This increase is caused by the fact that a lookup for existing records takes more time as more records are in the database. In order to get an indication of this growth, we added a trend line in Figure 9.4. We used Microsoft Excel to create a trend line. This trend line is described by Equation 9.3. The correlation coefficient(R^2) for Equation 9.3 is $R^2 = 0.79395$.

The histogram presented in Figure 9.5 shows the distribution of the execution times as well as the cumulative value. The latter value shows the percentage of actions with respect to the the total number of actions which are executed in less then the number of milliseconds on the x-axis. Looking at Figure 9.5 shows that 80% of the actions are stored in less than 100 milliseconds.

$$y = 0.0122x + 11.249 \tag{9.3}$$

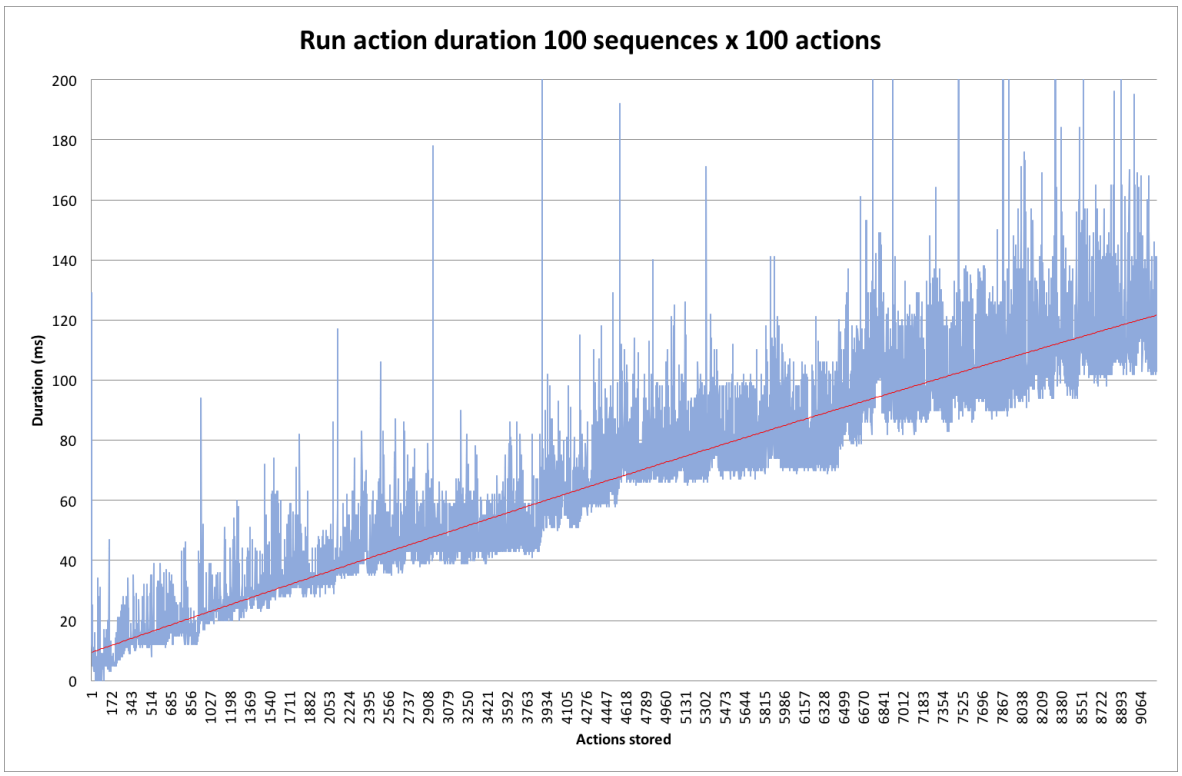


Figure 9.3: Duration in detail for storing an action in the database

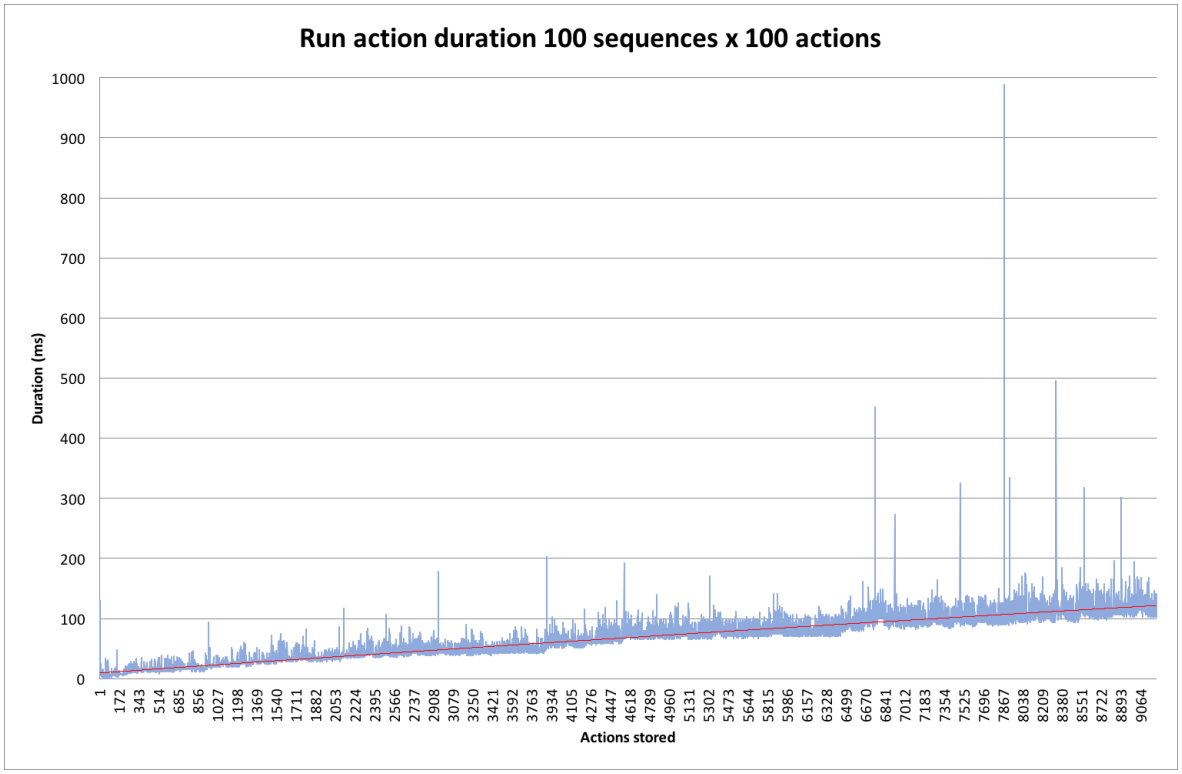


Figure 9.4: Duration for storing an action in the database

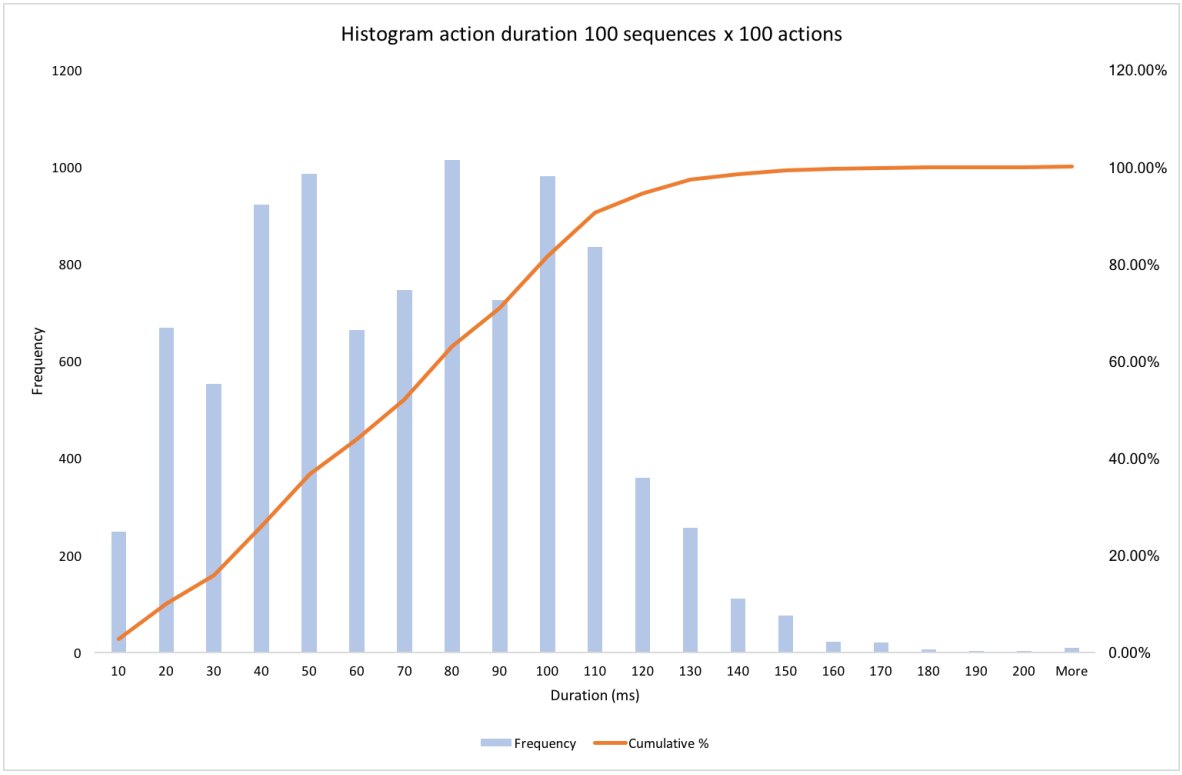


Figure 9.5: Histogram for the action duration for test with 10,000 samples

Storing a `State` Figure 9.6 shows execution time for each stored, or updated, state during the real-world benchmark with 10,000 samples. Comparing the data from Figure 9.6 with the result of the synthetic benchmark (see Table 9.10), shows that the synthetic benchmark provides a good prediction for duration of the code when executed during a run of TESTAR.

It is expected that the execution time will increase as more data is stored in the database. In order to get an indication of this growth, we added a trend line in Figure 9.6. This trend line is described by Equation 9.4. The execution time only grows 0.2 microseconds for each call to store a state. Due to big fluctuations in the data, the trend line has a low correlation coefficient of $R^2 = 0.002$. Due to the large outliers, it is not clear how the execution time changes by looking at Figure 9.6. We added Figure 9.7 to provide a better insight.

$$y = 0.0002x + 2.4326 \tag{9.4}$$

Looking at the histogram presented in Figure 9.8, shows that 95% of the states are stored within less than 10 milliseconds. It also show that storing a state takes at least more than 1 milliseconds. These values show up as zero's in the data. This is caused by the fact that the smallest time unit presented in the logging is 1 millisecond. This value belongs to a call where the state already exists in the database. The value is less than 1 milliseconds which could not be printed when the execution time was logged.

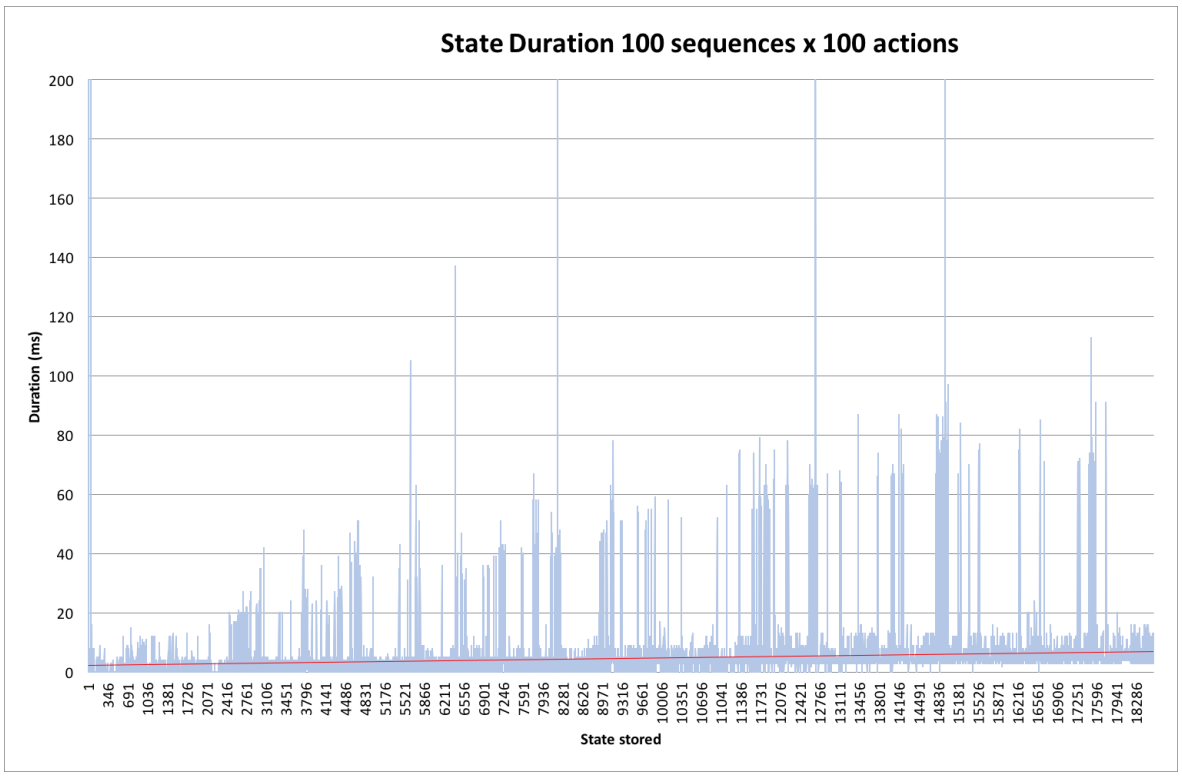


Figure 9.6: Duration for storing a state in the database

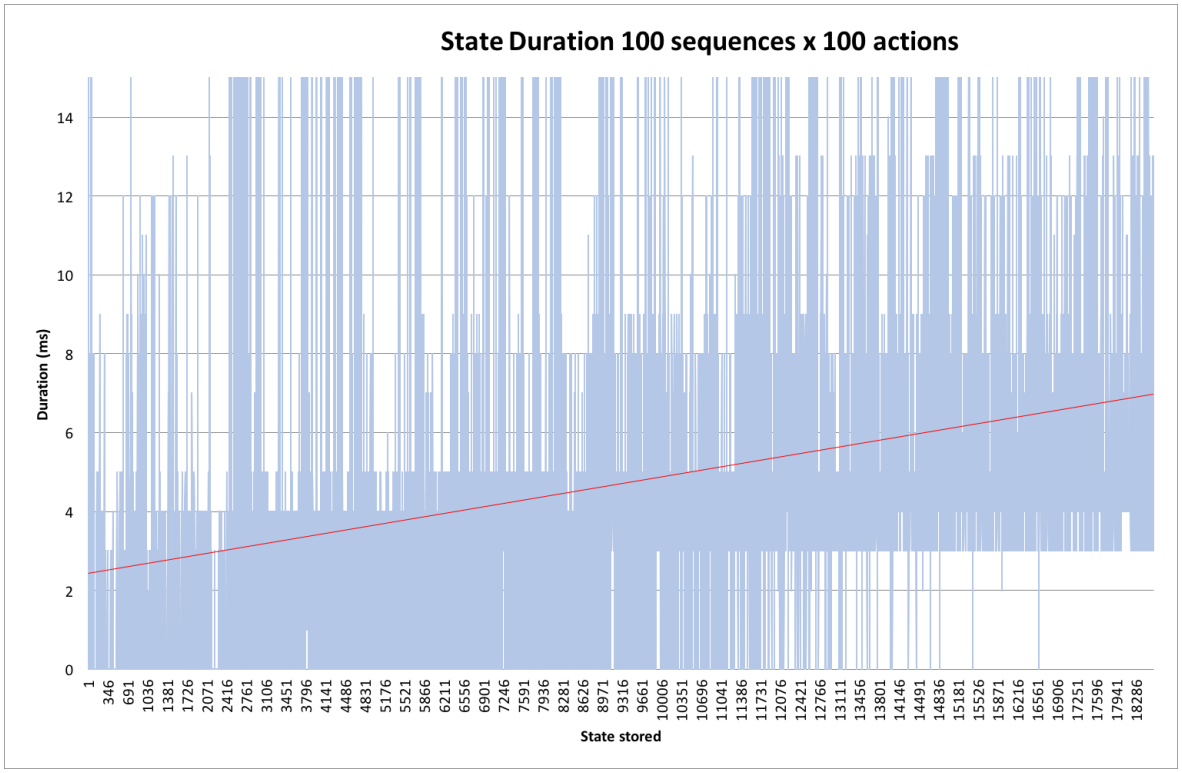


Figure 9.7: Duration in detail for storing a state in the database

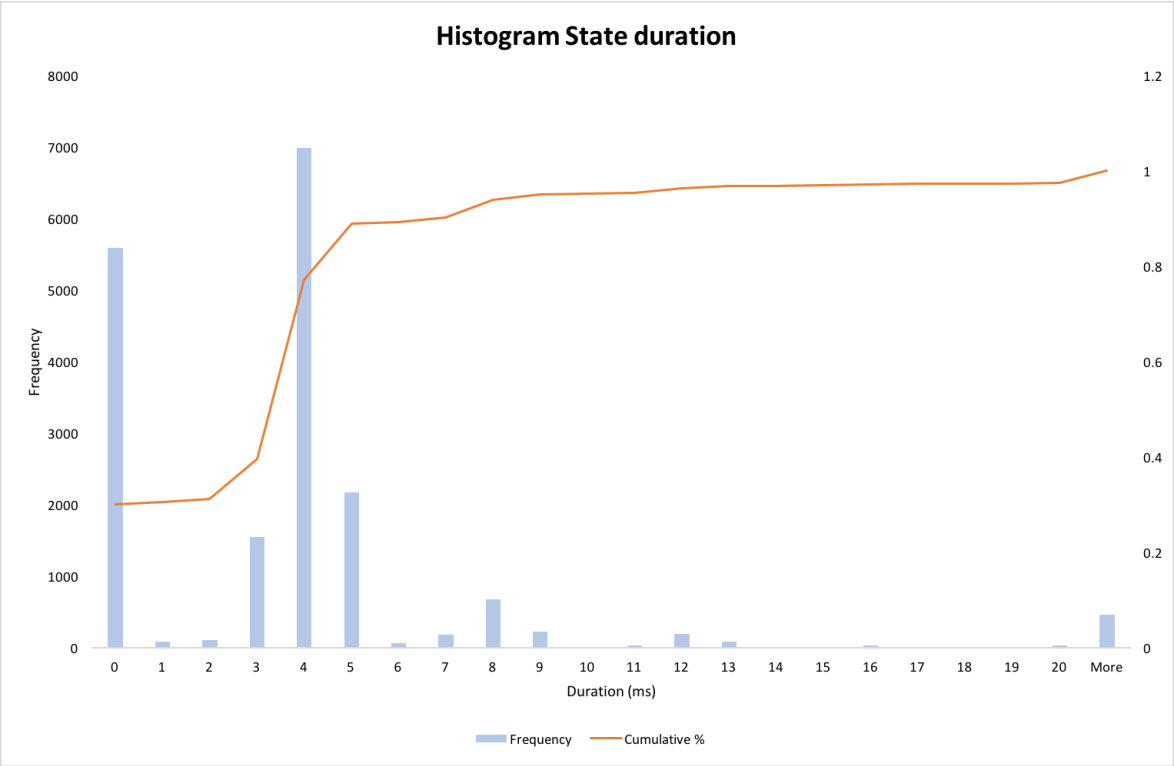


Figure 9.8: Histogram for the state duration for test with 10,000 samples

Storing a `Widget` Figure 9.9 shows execution time for each stored, or updated, widget during the real-world benchmark with 10,000 samples. Comparing the data from Figure 9.9 with the result of the synthetic benchmark (see Table 9.10), shows that synthetic benchmark provides a fair prediction for duration of the code when executed during a run of TESTAR.

It is expected that the execution time will increase as more data is stored in the database. In order to get an indication of this growth, we added a trend line in Figure 9.9. This trend line is described by Equation 9.5. The execution time only grows 1.2 microseconds for each call to store a widget. The trend line has a low correlation coefficient of $R^2 = 0.3763$.

$$y = 0.0012x + 3.083 \tag{9.5}$$

The histogram presented in Figure 9.11, shows that 80% of the widgets are stored within less than 20 milliseconds. It also shows that storing a widget takes at least more than 0 milliseconds . This value belongs to a call where the widget already exists in the database. The value is less than 1 milliseconds which could not be printed when the execution time was logged.

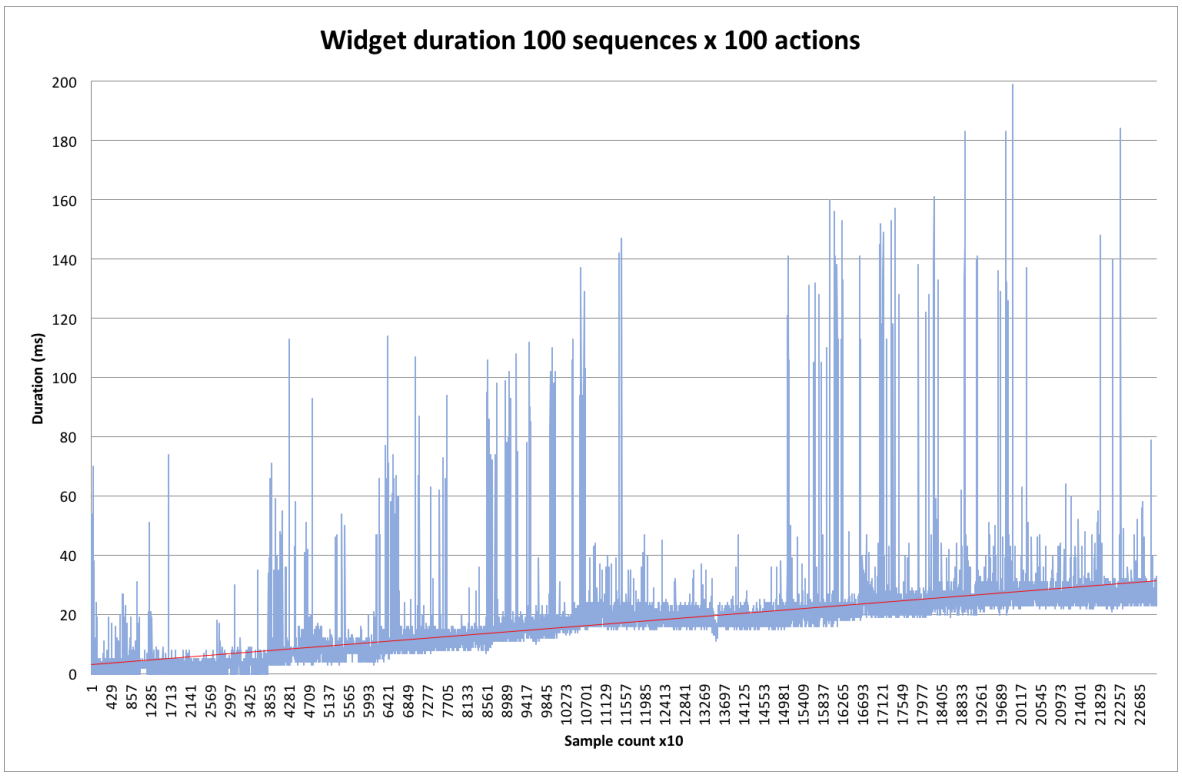


Figure 9.9: Duration for storing a Widget in the database

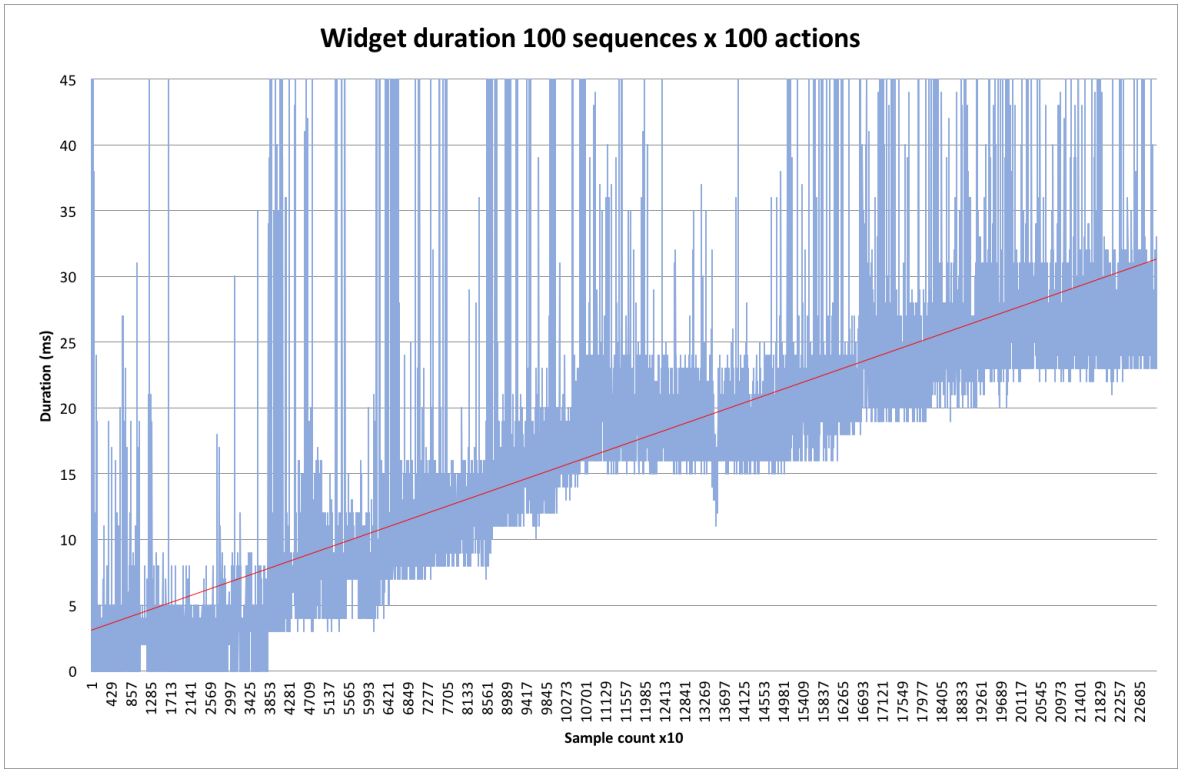


Figure 9.10: Duration in detail for storing a Widget in the database

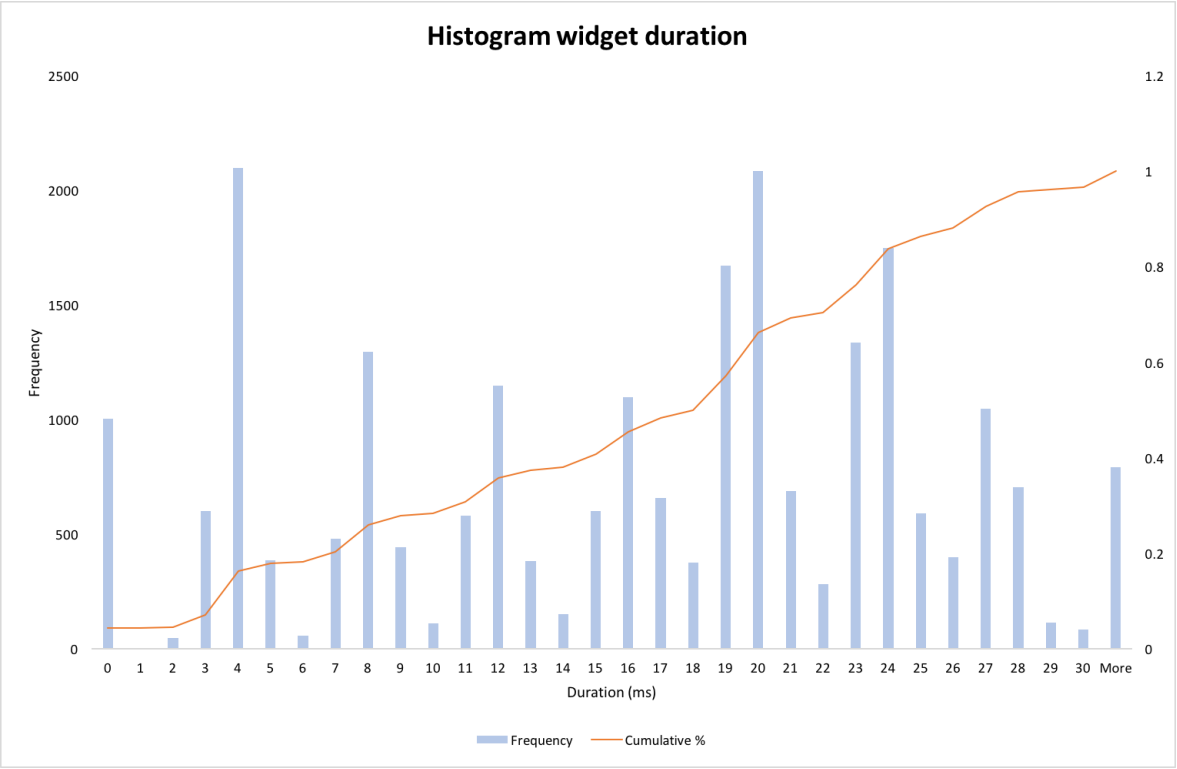


Figure 9.11: Histogram for the Widget duration for test with 10,000 samples

Artefact	Change in duration
State	0.2 microseconds (see Equation 9.4)
Action	12.2 microseconds (see Equation 9.3)
Widget	1.2 microseconds(see Equation 9.5)

Table 9.11: Change in execution time

TOTAL TEST EXECUTION TIME

During the execution of TESTAR, we measured the time it takes to execute a single action (the execution time of the method `run_action`). The same measurements are done for two scenarios. The first scenario executes the tests with the graph database enabled. In the second scenario, the graph database is disabled. Figure 9.12 shows the results of these measurements. Despite the fluctuation in the data, it is clear that an execution of `run_action` takes more time with the database enabled. It is also clear that the gap increases as more actions are executed.

As shown in Table 9.12 shows, the use of the database adds almost half an hour to the total execution time. The main reason for this increased execution time is the fact that all database operations occur on the thread of TESTAR.

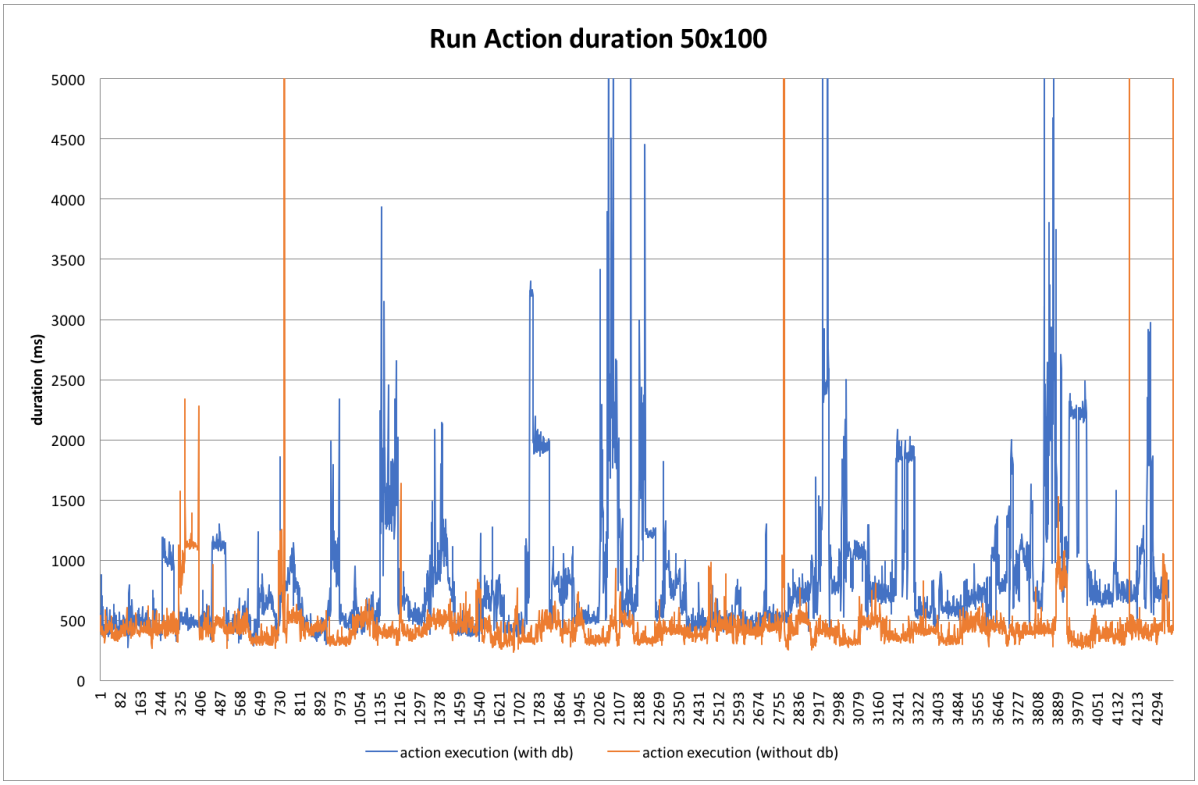


Figure 9.12: Duration of the "run action" step of TESTAR for a test with 5,000 actions

Scenario	Total execution time
5,000 actions with the database disabled	54 minutes and 4 seconds
5,000 actions with the database enabled	1 hour 22 minutes 49 seconds

Table 9.12: Total execution time for a test with 5,000 samples

REFLECTION ON EXECUTION TIME

Looking at the results, it is clear that the time required to store an artefact increases over time. Looking at the graphs presented in the section, it is clear that there is a lot of variation in the time required to store an artefact. Part of this fluctuation is caused by the fact that not every action results in writing data to the database. For instance, when an existing state is visited, no state data is written to the database. However, all the graphs show outliers where a database action takes more than 10 times the amount of time as the average for the same graph. We looked into the available data from the test and could not pinpoint a particular state or action which caused these outliers. Most likely, other activities in the environment that was used to execute the test could have influenced the the timing. The outliers in Figure 9.4 are an example of this behaviour. Another cause for the outliers could be found in the way OrientDB caches its data. OrientDB uses both read and write caches to optimize its performance. A cache hit will have an impact on the execution time of the database actions. To understand the exact timing behaviour, more research is required that is out of the scope for this work.

To conclude this section, an overview of the change in duration for the actions performed on the database is listed in Table 9.11. The change of duration for the millionth action takes ± twelve seconds more compared to the first action. As a run of five thousand actions takes ± one hour, it would require a test run of two hundred hours to get to the millionth action.

9.3.5. REQUIRED DISK SPACE FOR ORIENTDB

OrientDB uses the local file system to store the data. Knowledge about how this data is structured, is important to understand how disk space will grow as more data gets recorded. Table 9.13, lists the type of files that play a role in the discussion presented in this section.

Part	Extension	Description
Clusters	*.pcl *.cpm	Files that contain the data (*.pcl) and files containing the mapping between a record's cluster and the real physical location
Write ahead (operation) log	*.wal	Log file containing all storage actions used for recovery in case of a crash
Hash index	*.hib	Hashes for the functions implemented by OrientDB

Table 9.13: Important parts of the OrientDB data structure on disk (see [29] for more information)

It is expected that the disk space will have a certain minimum value to store the minimum required data for OrientDB to work. The usage will increase as more data is entered in the database. In order to see how the database will grow, we executed a series of TESTAR runs with an increasing number of action steps in each run.

Figure 9.13 and Table 9.15 present the results of the measurements which were measured running TESTAR against VLC. The artefact size information is retrieved through the database console (see [30]) command "LIST CLUSTERS". The size information retrieved through this command deviates from the total size on disk which was measured using standard tools from the operating system. The size on disk is much larger than the size of the artefacts as presented by OrientDB. One of the objects stored on disk which introduces this overhead is the *.wal file. This is the "write ahead log" (see [30]) which is a log file that is used for recovery when OrientDB crashes. The write ahead log can take up a large amount of data. Especially the database with the results of the run with 5,000 samples suffers from this overhead. This database was used extensively to perform queries. The folder created by the database always contains a file name `OFunction.name.hib`. Table 9.14 shows the results when we correct the disk usage for the known overhead.

Figure 9.13 shows the required disk space with respect to the number of artefacts stored. From this figure it looks as if the growth can assumed to be linear.

When we compare the disk usage per action from Table 9.15 to the same numbers used by the existing artefacts (see Table 9.16), we can conclude that the graph database only adds approximately 10%[11] to the growth of disk space per action.

[11] The overhead of the data model of OrientDB plays the smallest role in a test with a large number of actions. Therefore, we only looked at the test with 5,000 and 10,000 actions.

Scenario	OFunction.name .hib (kB)	*.wal (kB)	original disk usage (kB)	corrected disk usage (kB)
100	16,385	4,352	29,389	8,652
1,000	16,385	28,096	56,934	12,453
5,000	16,385	78,720	113,664	18,559
10,000	16,385	3,136	59,068	25,433

Table 9.14: Correction of disk space used by OrientDB

Scenario	Artefact size (kB)	corrected disk usage (kB)	Artefacts	Size per action (kB)
100	546	8,652	2,041	86.52
1,000	3,102	12,453	13,231	12.45
5,000	8,232	18,559	35,936	3.72
10,000	12,021	25,433	59,068	5.91

Table 9.15: Measured disk usage for TESTAR runs with an increasing number of actions

Scenario	Required disk space (MB)	Size per action(kB)
100	11.98	122.7
1,000	44.38	45,49
5,000	203.87	41.75
10,000	463,30	47,44

Table 9.16: Measured disk usage for the existing artefacts of TESTAR

REFLECTION ON DISK USAGE

In this section we analysed the disk usage of OrientDB when it is used to store states, actions and widgets from a TESTAR run. Without using the database, a run with TESTAR with 10,000 actions (steps) requires 47,44 kilo byte per action (see Table 9.16). This includes space required to store logs, sequences, screenshots, etcetera. Looking at the numbers for the test with 10,000 actions in Table 9.15, shows that storing data in a OrientDB graph database causes an increase in the usage of storage of 5,91 kilo bytes which equals 12,45% of the total storage requirement of TESTAR when a single action is stored.

In absolute numbers, OrientDB adds 5.9 kilobytes (numbers from Table 9.15) for each action. Theoretically this would mean over 80 million actions can be stored on a 500 gigabyte hard disk. Note, this theoretical exercise assumes the required disk space grows linear. The theoretic numbers will never be met in practical conditions since OrientDB will add an increase amount of overhead. With each write to the database, the write ahead log will also grow. Furthermore, the way OrientDB distributes it's data also adds to the overhead. In order to get a better measurement for the disk space, an experiment needs to be performed which fills the disk to it's maximum capacity.

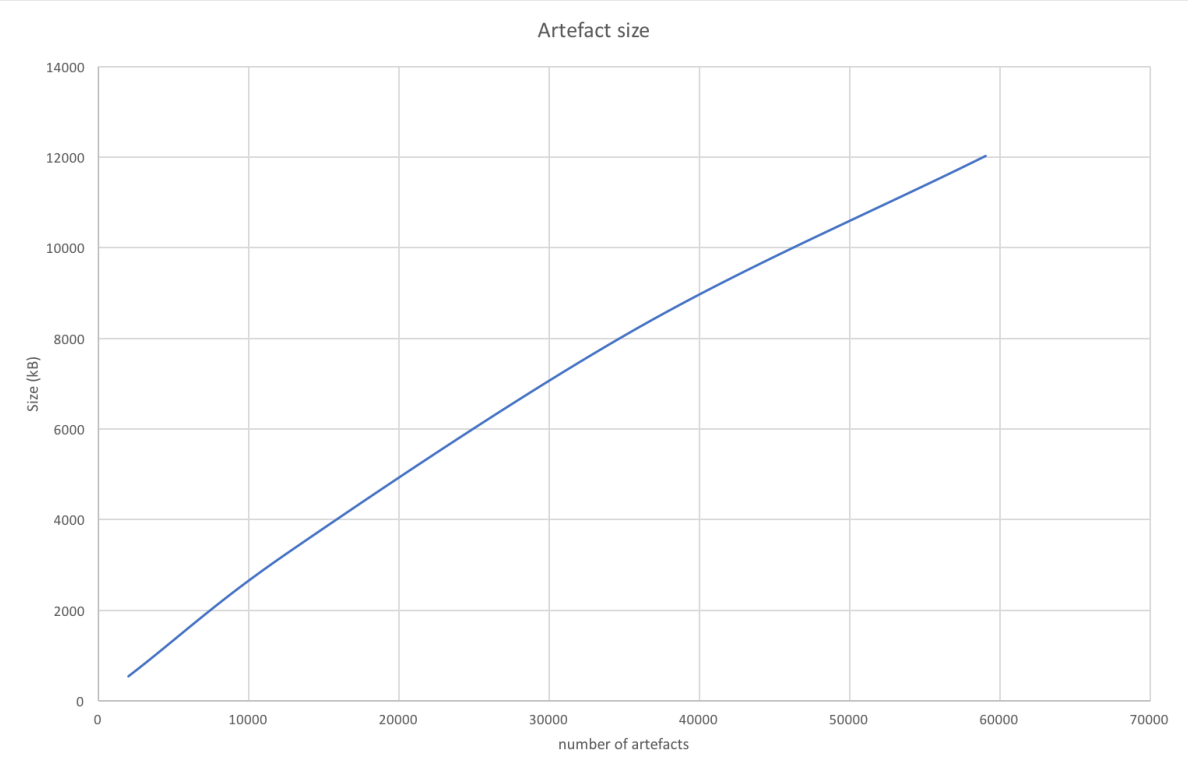


Figure 9.13: Disk usage versus the number of artefacts

9.3.6. CONCLUSION

In answer to question Q4, we can conclude that the introduction of the graph database adds between three and five kilobytes of data to the required disk space for TESTAR for each artefact (Edge or Vertex) stored in the database. A test with 10,000 action steps adds 12,000 kilobyte of data. The required disk size to store this data is even bigger. How big depends on a lot of factors. For instance, OrientDB also requires disk space for caches and indices.

The current solution also slows down the execution of TESTAR. On average, the duration of the execution of one `runAction` execution will increase from 454 milliseconds to 842 milliseconds[12]. The execution time of TESTAR can be improved when the database action will be executed from a separate thread. Currently, these action are performed on the same thread which is used to run the actions. These improvements will be left as future work.

The impact of the graph database on the execution time will change over time. We looked at this impact for the various type of artefacts stored by TESTAR. Depending on the type of artefact stored, the execution time increase between 0.2 microseconds (for a `State`) and 12.2 microseconds (for an `Action`).

[12]Data taken from a run with 5,000 actions

The results presented in Section 9.3.4 could be mapped to other applications since VLC is a representative application when it comes to complexity. Performance figures for other applications might deviate from the results presented in this research as other applications might consume less or more resources (memory, cpu-cycles). In the end, an application with the same complexity (number of menu options, dialogues, etc) will lead to the same number of artefacts stored in the database.

In our tests, we used the random algorithm for action selection. Looking at the results, this is not a very efficient way to walk through the menu's of the application. When another algorithm is selected where each action is triggered only once, the number of useless visits to the database, where no information is added will be less. This has to be validated in future research.

Using the graph database, introduces an overhead. In Table 9.12, we show that a run without using the graph database takes 28 minutes less compared to a similar run with the graph database enabled. As long as the SUT does not have any restriction on the maximum amount of time between two related actions, this overhead does not have to be a problem.

The calls to the graph database are performed on the same thread as the thread used by TESTAR to execute the action on the SUT. The design could be improved by delegating the database actions to a separate thread. This will be listed as future work. (see Section 10.1.1).

As a future improvement of the graph database implementation, the user should be able to select a subset of the available properties of an artefact. This would reduce the required disk space and the time required to store an artefact.

9.4. VALIDATING THE ADDED VALUE OF TESTAR TO MANUAL ACCESSIBILITY EVALUATION

In the accessibility part of the case study, we manually evaluated the accessibility of VLC and also applied TESTAR with the WCAG2ICT protocol on VLC in order to validate the added value that TESTAR provides to experts. The manual evaluation took place after the evaluation runs with TESTAR, so that our validation of the evaluation results from TESTAR was unbiased by any manual observations. We present the manual evaluation first so that it can serve as a reference point for what follows. We then present our results with TESTAR and conclude by comparing TESTAR to the manual evaluation.

In this section, we answer the following research question:
How do experts benefit from our implementation of accessibility evaluation in TESTAR? (Q9)
This question has the following sub-questions:

1. What violations can we find in VLC with a manual evaluation? (Section 9.4.1)
2. How does the sequence length affect the number of violations that TESTAR finds? (Section 9.4.2)
3. How much effort is needed to verify the warnings that TESTAR finds? (Section 9.4.2)
4. How reliable are the errors that TESTAR finds? (Section 9.4.2)
5. How easy to use and readable is the evaluation report from TESTAR? (Section 9.4.2)
6. How do the evaluation results from TESTAR compare to the results from the manual evaluation? (Section 9.4.3)

9.4.1. MANUAL ACCESSIBILITY EVALUATION

In this section we present the results from a manual accessibility evaluation of VLC. The remainder of the case study will then compare our results with those from TESTAR.

METHOD

In practice, experts can define and evaluate one or more key processes in a system in order to structure the accessibility evaluation around essential functionality that users will want to perform. However, it is also important to select a structured sample from the system to ensure a thorough evaluation, as well as to select a random sample to verify that the structured sample is adequate. In broad terms, this is the approach suggested in Website Accessibility Conformance Evaluation Methodology (WCAG-EM) 1.0 [38], an evaluation methodology for web accessibility that accompanies WCAG 2.0.

We chose to define and evaluate five key processes that cover typical use of a media player. This approach exploits the knowledge an expert has compared to the random action selection algorithm in TESTAR. Since the goal was not to perform an exhaustive accessibility evaluation, we did not select further structured or random samples. We believe that the five processes lead to enough different areas of VLC to compare the results to those from TESTAR, which did not explore VLC very deeply either. The manual evaluation is restricted to all level A WCAG2ICT success criteria, which includes non-automatable rules not covered by TESTAR.

The five processes that we chose to manually evaluate in VLC are:

1. Opening a media file and starting playback
2. Pausing and resuming playback
3. Skipping in a media file
4. Getting information about a media file
5. Changing preferences

For the evaluation we used the free, open source Windows screen reader NonVisual Desktop Access (NVDA)[13] version 2017.4. Screen readers are assistive technology that allows blind and visually impaired people to use computers. Screen readers read, braille and/or magnify the screen contents. Using a screen reader for the evaluation means that it is somewhat biased towards accessibility for blind and visually impaired users, but it has the advantage that a screen reader uses the same accessibility APIs as TESTAR to obtain information about the SUT.

Because TESTAR only uses keyboard actions during accessibility evaluation, we likewise used only the keyboard to interact with VLC. As was the case when we restricted TESTAR to keyboard actions, interacting with VLC through the keyboard alone allows us to include keyboard accessibility (guideline 2.1) in the evaluation. Furthermore, screen reader users typically will not use the mouse since they cannot see the mouse pointer very well, if at all. Windows has many default shortcut keys, including `Tab` and `Shift+Tab` to navigate between widgets, `Enter` and `Space` to interact with a widget and the arrow keys to move around in a widget. The GUI of VLC also contains shortcut keys, but they did not always work. Therefore, we also consulted the VLC hot keys table[14].

Table 9.17 shows the errors that we found for each process when manually evaluating the accessibility of VLC and the time that each process took to evaluate. Table 9.18 also presents the number of errors, but then distributed over the related WCAG2ICT success criteria. Since verification is inherent to a manual evaluation, we do not have warnings and can therefore speak of errors rather than violations.

[13] https://www.nvaccess.org/
[14] https://wiki.videolan.org/Hotkeys_table/
[15] Rounded up to 5 minutes.

Process	Description	Evaluation Time[15]	Errors
1	Open and play	5 minutes	0
2	Pause and resume	15 minutes	5
3	Skip	5 minutes	2
4	Get information	10 minutes	5
5	Change preferences	15 minutes	6
Total		50 minutes	18

Table 9.17: Errors found and time spent in manual accessibility evaluation of VLC (ordered by process)

Criterion	Description	Errors
2.1.1	Keyboard	9
2.1.2	No Keyboard Trap	1
3.3.2	Labels or Instructions	1
4.1.2	Name, Role, Value	7
Total		18

Table 9.18: Errors found in manual accessibility evaluation of VLC (ordered by WCAG2ICT success criterion)

In total we found 18 errors and spent approximately 50 minutes, so that the average time to find an error is approximately 2.78 minutes. The rate of errors found per minute depends on the processes. For example, we found no errors in the first 5 minutes because we began with process 1, but we would have found an error approximately every 2.5 minutes if we had started with process 5.

The rest of this section presents the manual evaluation results for each process. WCAG-EM, which uses the terms evaluation findings and evaluation results interchangeably, suggests to begin by reporting the outcomes of each prior step in the methodology. It also suggests optional report items, including an aggregated score. The mandatory content basically follows the structure of the methodology, that comprises five steps including the reporting step. A full discussion of the methodology is beyond the scope of this case study, since the goal is a comparison with TESTAR and not an exhaustive and accountable expert evaluation of VLC. We only used two steps from WCAG-EM to perform the evaluation: Include Complete Processes (step 3.c) and Check All Complete Processes (step 4.b). We do not claim to be proficient with WCAG-EM as a whole, but having extensively studied WCAG2ICT to implement its rules in TESTAR gives us enough confidence to perform this evaluation in the context of a case study.

PROCESS: OPENING A MEDIA FILE AND STARTING PLAYBACK

This process was both quick and fully accessible. The first menu on the menu bar is "Media". In that menu, "Open File" is the first item and has a shortcut key, `Ctrl+O`, that is familiar from other Windows software. Activating the menu item opens a dialogue to select a media file. The dialogue is a standard, accessible Windows dialogue. Pressing `Enter` on a file opens it and playback starts automatically. We found no errors.

PROCESS: PAUSING AND RESUMING PLAYBACK

This process took longer because the shortcut keys we found in VLC did to play and pause not work (success criterion 2.1.1: Keyboard). The VLC hot keys table contained other hot keys that did work. The process is accessible when using the hot keys table.

The "Play", "Pause" and "Stop" menu items are available from the "Playback" menu on the menu bar. `Space` toggles between play and pause and the `S` key stops playback. These are the hot keys from the VLC hot keys table.

We found the following errors related to success criterion 2.1.1: Keyboard:

- The main window contains widgets to play, pause and stop playback, but they cannot be reached with `Tab` and `Shift+Tab`.
- "Play", "Pause" and "Stop" are available from the context menu that appears when right-clicking in the main window. The `Application` key is the standard shortcut key to open a context menu in Windows, but it does not work in VLC, so the context menu is not accessible with the keyboard.
- The menu items to play and stop found in the "Playback" menu have shortcut keys. `Alt+P` should play and `Alt+S` should stop, but they do not work in the main window.
- The menu item to pause found in the "Playback" menu has no shortcut key.
- The shortcut keys from the VLC hot keys table to toggle between play and pause and the hot key to stop work. The table also includes the `[` key to pause only and the `]` key to play only, instead of toggling playback, but they actually changed the playback speed.

PROCESS: SKIPPING IN A MEDIA FILE

This is another process where there are two sets of shortcut keys, one from the GUI of VLC and one from the VLC hot keys table, where the set from the GUI did not work. Again, the process is accessible with the shortcut keys from the hot keys table, assuming the user is satisfied with the default jump lengths that VLC provides. It is also possible to skip to a specific time within the media file, but we did not evaluate this.

The "Jump Forward" and "Jump Backward" menu items are available from the "Playback" menu on the menu bar. Shortcut keys are available to perform jumps of different

lengths based on combinations of `Shift`, `Alt` and `Ctrl` with the left and right arrow keys. These are the shortcut keys from the VLC hot keys table.

We found the following errors related to success criterion 2.1.1: Keyboard:

- The main window contains widgets to skip forward and backward, but they cannot be reached with `Tab` and `Shift+Tab`.
- The menu items to jump forward and backward found in the "Playback" menu have shortcut keys. `Alt+J` should jump forward and `Alt+K` should jump backward, but they do not work in the main window.

PROCESS: GETTING INFORMATION ABOUT A MEDIA FILE

There are multiple sources of information about a media file in VLC. The main window has a title bar with the artist and title of the media file. VLC also has a status bar that can be enabled via the "View" menu and choosing "Status Bar". By default, the status bar includes the elapsed and remaining playback time. For more information, there is a dedicated dialogue under the menu item "Media Information" in the "Tools" menu on the menu bar or with shortcut key `Ctrl+I`. The title bar and status bar are fully accessible, but the media information dialogue posed some difficulties.

We found the following error for success criterion 2.1.1: Keyboard:

- The "Statistics" tab of the "Media Information" dialogue contains a tree view. Keyboard focus does not consistently return to the same tree view item when navigating through the dialogue with `Tab` and `Shift+Tab`.

We found the following error for success criterion 2.1.2: No Keyboard Trap:

- Among the information on the "General" tab of the "Media Information" dialogue there is a "Comments" field. After keyboard focus enters this widget, it is impossible to leave it with `Tab` and `Shift+Tab`, nor by using `Ctrl+Tab` to switch to another tab. It is possible to leave the widget with `Escape`, but then the whole dialogue closes.

We found the following errors for success criterion 4.1.2: Name, Role, Value:

- NVDA does not announce the labels for the fields on the "General" tab of the "Media Information" dialogue. This tab also allows editing the information, meaning that there are input widgets and labels describing what information to enter. The labels do not appear to be associated with their input widgets, so NVDA only reads the value of an input widget, such as "2010", but not its label, such as "Date".
- The information on the "Meta data" and "Statistics" tabs of the "Media Information" dialogue is structured by a tree view. NVDA reads the name of the tree view items, but not the value that contains the actual information. We count one error per tab, so two errors total.

PROCESS: CHANGING PREFERENCES

VLC is very customizable. The user can customize the GUI, choose a different interface altogether, manage playback effects and more. We only evaluated the dialogue with preferences accessed through the "Preferences" menu item in the "Tools" menu on the menu bar or through the shortcut key `Ctrl+P`. Moreover, under "Show settings" we left the radio button to select the preferences view at its default to show the "Simple" view.

We found the following error for success criterion 2.1.1: Keyboard:

- Keyboard focus can land on the pane containing the preferences. A pane is typically a presentational, non-content widget that cannot be interacted with, so it should not receive keyboard focus.

We found the following error for success criterion 3.3.2: Labels or Instructions:

- Groups of preferences and individual preferences have a description. However, NVDA announces HTML tags along with the description text for many of these descriptions, for example for the preference "Show controls in full screen mode". Formatting code from the GUI seems to incorrectly end up in the accessibility API.

We found the following errors for success criterion 4.1.2: Name, Role, Value:

- The preferences tabs have the check-box role, according to what NVDA announces, even though they cannot be checked or unchecked. This is likely how VLC indicates which tab is active, but it causes NVDA to always announce the active tab as a checked check-box.
- NVDA does not announce the new value when using the up and down arrow keys to select from a list of values, for example for the "Language" preference.
- NVDA does not announce the label for the search field on the "Hotkeys" tab. The label does not appear to be associated with the search widget.
- The hot key assignments on the "Hotkeys" tab are structured by a tree view. NVDA reads the name of the tree view items, but not the value that contains the hot key.

9.4.2. ACCESSIBILITY EVALUATION WITH TESTAR

In this section we present our results from an automated accessibility evaluation of VLC with TESTAR. We set up TESTAR to use the random action selection algorithm and sequence lengths varying between 50 and 250 actions. Appendix G lists the settings we used in TESTAR.

The sequence length was determined by trial-and-error. We established the upper limit after deciding to perform 5 runs and then determining the linear growth rate for the

sequence length. The initial growth rate was 500 actions, meaning that the sequence length would vary between $1 * 500 = 500$ and $5 * 500 = 2500$ actions. However, the run with 500 actions that we performed already took approximately 10 minutes and yielded an evaluation report that was too long to extensively validate for this case study. A second run with the same sequence length took less time and produced a considerably shorter evaluation report. This is due to the random action selection algorithm, which is more likely to select different actions and therefore visit more states if the sequence is long, but that will not do so all the time. Also, when the sequence is too short, there is not enough opportunity to get out of the initial state. So we eventually set the growth rate to 50 actions and aim to validate if the resulting sequence length results in enough information.

This section starts with an excerpt from the evaluation report generated by TESTAR for VLC, with the WCAG2ICT protocol and graph database support enabled. The excerpt shows part of the evaluation results for the initial state in the 50-action sequence. Because the starting point was the same for all runs, all resulting evaluation reports begin in the same way. They diverge when the random action selection algorithm causes TESTAR to enter different states.

We then validate the results from TESTAR on quantitative and qualitative aspects. The first topic is how varying the sequence length affects the number of violations. We already found that the sequence length is an insufficient indicator of the amount of results, but it will have an effect on the duration of the test. The sequence length is therefore a useful test setting and we discuss how it affected our results. The second topic is the quality of the violations that TESTAR found. To estimate the effort for an expert to use TESTAR, we recorded how long it took to verify the warnings that TESTAR found. Since errors are defined as definite violations (see Section 2.4) we also verified the errors, not something an expert should have to do, to assess their reliability. Having carefully perused the evaluation report by this time, we end the section by suggesting ways to improve its ease of use and readability.

VLC EVALUATION REPORT FROM TESTAR

Accessibility Evaluation Report

General information

- Report time: 2018-01-21 17:42:45
- Report type: GraphDB
- Accessibility standard implementation: WCAG2ICT-20171121
- Sequence number: 1

States with violations

Unique states with violations: 6

Type	Count
Error	20
Warning	2
Pass	76
Total	98

State: SC1o5dzzy307482039866

Screenshot

Open screenshot in a new window

Violations

Type	Criterion	Level	Widget	Message
WARNING	1.1.1 Non-text Content	A	Possible missing text alternative	
WARNING	2.1.2 No Keyboard Trap	A	N/A	Possible widgets missing shortcut keys
ERROR	3.1.1 Language of Page	A	VLC media player	Missing top-level language identifier
…	…	…	…	…

Off-line evaluation

General information

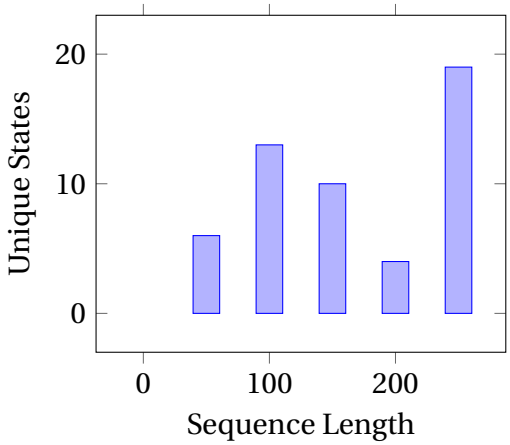
Type	Count
Error	0
Warning	0
Pass	2
Total	2

Violations

None

VALIDATING THE NUMBER OF VIOLATIONS

Unique states with violations for different sequence lengths



Total violations for different sequence lengths

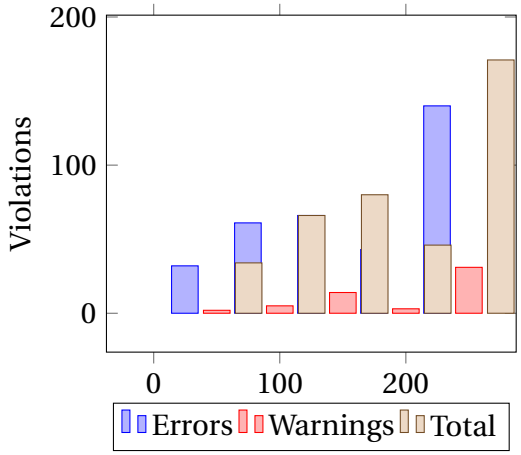


Figure 9.4.2 shows the number of unique states with violations for different sequence lengths. The runs with 100 and 200 actions had no states without violations and the runs with 50, 150 and 250 actions had one state without violations. Moreover, the number of states and the number of states with violations do not relate to the sequence length. From the data it appears that increasing the sequence length is an unreliable way to find more states with violations. Furthermore, if the violations in the initial state are resolved, TESTAR will waste actions until it reaches a new unique state. This inefficiency will continue to grow as violations in states reachable from violation-free states are also resolved. A depth-first action selection algorithm could lead to better performance. Alternatively, the test protocol could be adapted to a specific SUT to start testing on a different state.

Compared to the sequence length, there is a low number of unique states. The reason is that we used the random action selection algorithm. At worst, this algorithm always selects an action that does not lead to a new unique state, meaning that TESTAR enters the first unique state when it starts and then stays there. At best, the algorithm selects only actions that enter new unique states, meaning that the number of unique states is equal to the sequence length. Being random, the actual number cannot be predicted. The number

also depends on how TESTAR defines uniqueness. For example, a different run with 500 actions yielded 277 unique states, 268 with violations. However, many of these states had a constantly updating time display widget in them that caused TESTAR to treat them as unique even though the rest of the state did not change. Only reporting unique states with violations helps to cut down clutter in the report, but uniqueness does not guarantee that there is no redundancy.

Next, we look at the total number of violations. Figure 9.4.2 shows the number of errors and warnings for different sequence lengths. Again the effects of the random action selection algorithm are apparent. Increasing the sequence length increases the chance of finding more violations, since more actions are executed, but the run with 200 actions shows that this does not always happen. It had fewer violations than the run with 150 actions (46 actions and 80 actions, respectively). Reaching more unique states also does not imply finding more violations. The run with 150 actions had fewer unique states with violations than the one with 100 actions (10 and 13 unique states with violations, respectively), but had more violations (80 violations and 66 violations, respectively).

Since warnings require verification by an expert while errors do not, the number of warnings can be used to estimate how much work an expert has to perform. The longest run had 31 warnings. Assuming the evaluation report together with the graph database contains all the information an expert needs, these can probably be verified within an hour. We will test this assumption when we look at the quality of the evaluation results.

The conclusion from the validation thus far is that the sequence length is not a reliable stop criterion. The number of warnings could be a better stop criterion than the sequence length, because it can be used to produce a time estimate for the expert. However, other stop criteria such as time limits or SUT coverage may also be needed.

VALIDATING THE QUALITY OF THE EVALUATION RESULTS

To validate the quality of the evaluation results, we chose the evaluation report from the run with 150 actions. With 66 errors and 14 warnings (80 violations), this provides a good balance between report length and detail. The run took 131 seconds, counted from the first to the last timestamp in the debug log file. On average, this means that TESTAR found a violation every 1.64 seconds, but this number will not grow linearly with the test duration or sequence length because we used the random action selection algorithm. While the numbers look impressive, we are also interested in qualitative aspects.

TESTAR did not find any violations from off-line oracles during the case study. Since there are only two off-line oracles, this was not unexpected. The graph database has another important function for accessibility evaluation, however, namely when gathering information about violations from on-the-fly evaluation. The method we use throughout this research is Gremlin (see Section 3.4.2), particularly the Gremlin console (see Appendix E). Gremlin can be used to obtain all the widget and state properties. We made extensive use of this second function of the graph database while verifying warnings.

There are two other information sources that an expert can use to verify warnings,

making a total of three: the evaluation report, the graph database and the SUT itself. We used all three side-by-side. All warnings took less than 2 minutes to verify. To reach this efficiency, the expert must be familiar with querying the graph database with Gremlin.

There were only 2 oracles that yielded warnings:

- Non-text Content (success criterion 1.1.1, see Section 7.4.1): 11 warnings
- Keyboard (success criterion 2.1.1, see Section 7.4.2): 3 warnings

The non-text content warnings signal that there are images that have no text alternative. The images cannot be interacted with, otherwise the warnings would be errors. The 11 image widgets that caused warnings were all uniquely stored in the graph database, so we could query for their location and dimensions and locate them in the SUT using this informataion.

All but one non-text content warning occurred in a standard Windows dialogue to open media files. Such dialogues are not part of the SUT, but of the operating system. These dialogues were excluded from the test using the filtering engine that is built into TESTAR, but it is hard to eliminate them completely because there are multiple ways to reach them that only become apparent when TESTAR finds one not covered by a filter. A general solution not specific to VLC is difficult because the dialogues range from opening and saving files to printing and do not necessarily share a common property to distinguish them from the SUT dialogues.

The remaining warning was for an image in the main window of VLC. We verified it to be an error, because the widget is the “Mute” control. It should actually be possible to interact with it using the keyboard (see Section 9.4.1), but that is a separate violation that TESTAR did not report. Regardless of keyboard interaction, a distinction that we added to structure the evaluation results, success criterion 1..1 clearly states that controls require a name that describes their purpose (see Section 7.4.1).

The keyboard warnings signal that TESTAR found few shortcut keys in a state relative to the number of widgets. Using the SUT, it was easy to verify these warnings. 1 warning occurred in the main window. Indeed, many widgets in that window do not have shortcut keys, but menu items that perform the same commands in VLC often have shortcut keys. The low number of widgets with shortcut keys in the main window is therefore not an error. However, when verifying this warning we found that keyboard focus cannot be moved through the widgets, mostly playback controls, in the main window. Our manual evaluation revealed this too (see Section 9.4.1). So even though we verified that there are enough shortcut keys in the menus, verifying the warning revealed a related violation that is actually an error. The remaining 2 warnings occurred in the same standard Windows dialogue that had the text alternative warnings and are therefore outside the SUT.

The appearance of warnings and errors in standard Windows dialogues and the desire to exclude them requires further configuration of TESTAR before the automated evaluation can begin. This means more effort for a developer or possibly the expert to set up TESTAR,

for example with filters or black-lists. The classification framework for automated usability evaluation of user interfaces presented in Section 6.3 classified TESTAR as a guideline-based inspection tool for analysis of user interfaces. The *effort level* was specified as requiring minimal effort, but can now be more precisely described as configuring TESTAR for a specific SUT. The comes closest effort level in the framework, other than minimal, is model development [22]. However, model development suggests that TESTAR needs to be scripted to work with the SUT, whereas no effort or only simple configuration is actually required. After all, an important feature of TESTAR is that it explores the SUT on its own. So even though we signaled the need to spend some effort, the minimal *effort level* still seems to be the appropriate classification.

To discover potential false positives among the errors, we used the same three information sources as with the warnings. The evaluation report contained errors from the following oracles:

- Language of Page (success criterion 3.1.1, see Section 7.4.4): 1 error
- Name, Role, Value (success criterion 4.1.2, see Section 7.4.6):
  - Unknown role: 1 error
  - Missing name: 64 errors

The language error is valid, albeit not of much consequence as long as the SUT language and the operating system language are the same. The unknown role error is valid too. It was triggered by the role `UIAWindow` on the VLC main window. This is a false positive, because `UIAWindow` is the appropriate role for SUT windows. The oracle should not have yielded a violation for this role. We investigated how the `UIAUnknown` role was defined in TESTAR. This revealed a bug that we could easily resolve.

By far the most reported error is for widgets without a name. In the implementation in TESTAR, where we treat the name of an image as its text alternative, success criterion 4.1.2 is a generalization of success criterion 1.1.1. To avoid reporting the same widget twice, the oracle for missing names excludes images. Not counting images, there were 43 unique widgets without a name in the graph database. This number is lower than the 64 errors for this oracle because a widget that occurs in multiple states is stored in the graph database only once.

Over half of the missing name errors fell outside the SUT or were false positives. Querying the graph database for the related widgets showed that 26 were in standard Windows dialogues, which are outside the SUT. 12 missing names inside the SUT appeared on presentational, non-content widgets. These widgets had the role `UIAPane` or `UIASeparator`. We tried to ignore non-content widgets in the oracles, but this needs more real-world testing and refinement. The remaining 26 widgets with missing names were valid errors. Overall, this means that TESTAR found violations that we could report to a developer as errors to be resolved, but that TESTAR needs to be improved and that the evaluation results will require careful verification for now.

We also looked at the ease of use and readability of the evaluation report. The order of violations in the report is based on the evaluation order in TESTAR. In our implementation of WCAG2ICT, evaluation recursively goes through principles, guidelines and success criteria in a depth-first manner (see Section 8.3). This leads to a report in the same order as WCAG2ICT. It would be useful to separate errors and warnings so that either one could be hidden to focus on resolving errors or verifying warnings.

In the evaluation report, widgets are identified by their title. There are two oracles where this is problematic: the oracle that finds images without a text alternative (success criterion 1.1.1) and the more generic oracle that finds widgets without a name (success criterion 4.1.2). These widgets do not have a title, so a better way has to be found to identify them in the report. They could be highlighted in the state screenshot, but this is not yet implemented. Of course, the concrete ID could also be stored in the evaluation report to allow an expert to use it in graph database queries or with the Spy mode in TESTAR.

Finally, filtering the states by uniqueness before reporting them prevents duplicate evaluation results. This is possible thanks to the graph database. When graph database support is disabled, each state will be reported as it is evaluated during on-the-fly evaluation. This could lead to a much longer evaluation report. In this way the graph database significantly improves the readability of the report.

9.4.3. CONCLUSION

Criterion	Description	Total	False Positives	Outside SUT	Verified Errors
1.1.1	Non-text Content	11	0	10	1
2.1.1	Keyboard	3	0	2	1
2.1.2	No Keyboard Trap	0	0	0	0
3.1.1	Language of Page	1	0	0	1
4.1.2	Name, Role, Value	65	12	26	27
Total		80	12	38	30

Table 9.19: Violations found in accessibility evaluation of VLC with TESTAR (150 actions)

Criterion	Description	Verified TESTAR Errors	Manual Errors
1.1.1	Non-text Content	1	0
2.1.1	Keyboard	1	9
2.1.2	No Keyboard Trap	0	1
3.1.1	Language of Page	1	0
3.3.2	Labels or Instructions[16]	N/A	1
4.1.2	Name, Role, Value	27	7
Total		30	18

Table 9.20: Errors found in manual accessibility evaluation of VLC compared to TESTAR (150 actions)

[16] Not evaluated with TESTAR.

Table 9.19 gives insight into the number of violations that TESTAR found for VLC. The **Total** column shows the total number of violations. The **False Positives** and **Outside SUT** columns show how many violations, both warnings and errors, were false positives (inside the SUT) or were outside the SUT altogether. The **Verified Errors** are the remaining errors after subtracting false positives and violations outside the SUT from the total. Table 9.20 compares the number of verified errors from TESTAR to the number of errors that we found when manually evaluating VLC. Both tables only show WCAG2ICT guidelines for which we and/or TESTAR found results.

The tables show that TESTAR found more violations than we, but also that 62.5% were false positives or violations outside the SUT. By far the most violations that TESTAR found are for Name, Role, Value (success criterion 4.1.2). We also found a relatively high number of violations for this success criterion, but found more violations for Keyboard (success criterion 2.1.1). A substantial difference lies in the net violations, 100% of our results compared to 37.5% from TESTAR, since we could immediately skip writing down anything that would be a false positive or a violation outside the SUT. So looking at the numbers, TESTAR finds many violations, but at the cost of generating more noise.

It is clear that TESTAR outperformed us by finding 80 violations (30 verified errors) in just over 2 minutes compared to our 18 violations in 50 minutes. Of course, we also spent at most 2 minutes to verify each warning that TESTAR found, so less than 28 minutes for the 14 warnings. We deliberately do not include time to verify errors, because by definition this is not required for errors. Still, anyone developing oracles that yield errors must verify that they do not result in false positives or violations outside the SUT. This case study was an important opportunity to do that ourselves.

To summarize, with our help TESTAR found 30 verified errors in half an hour, an error each minute on average. However, a qualitative comparison tells a different story. TESTAR can only evaluate certain accessibility rules, whereas an expert can evaluate all the rules in an accessibility standard. This means that TESTAR can perform quite well, but only when complementing an expert. On one hand, TESTAR is effective at finding errors for rules that it can unambiguously evaluate, especially rules with simple on-the-fly oracles. It struggles a bit more with more advanced rules, but there is clearly a lot of potential here, not least in off-line evaluation. On the other hand, TESTAR did not find some errors that were immediately apparent during manual evaluation. Good examples are the inability to navigate through widgets in the main window of VLC using the keyboard and the context menu that can only be opened using the mouse. TESTAR can support an expert in finding technical violations such as widgets without a name, where an expert may miss a few widgets if there are hundreds to evaluate. Such technical evaluation tools for websites have existed for years and now TESTAR brings similar features to the desktop.

10

Conclusion

In this report we presented the results of our research. We investigated the benefits of storing the output of TESTAR in a graph database. Furthermore, we investigated how TESTAR can be used to evaluate the accessibility of desktop software. Using the features of TESTAR and the added capabilities of the graph database, we demonstrated that TESTAR can be used to assist an expert in performing an accessibility evaluation.

Our research demonstrated that adding capabilities to store data in a graph database provides flexibility to the user in terms of what information is stored in a test. One example of such a benefit is the fact that the new way of storing results provides the opportunity for off-line analysis. This is another feature we introduced during our research. Figure 10.1 shows the new execution diagram for TESTAR. In the figure, the interaction with the graph database is highlighted.

In Section 10.1 and 10.2 we elaborate on the research results from Floren and Davy, respectively. We include pointers for possible future research.

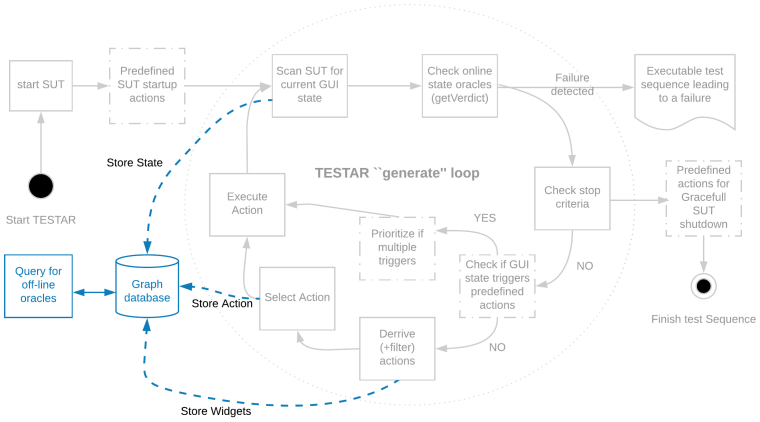


Figure 10.1: Updated TESTAR execution flow containing the interaction with the graph database

10.1. GRAPH DATABASE EXTENSION

In this section, we conclude on the research performed by Floren de Gier. Floren investigated the suitability of graph databases for storing test results, and implemented a corresponding extension in TESTAR (see Chapter 4 and 5). We used the database to store the States, Widgets and Actions during the execution of a test with TESTAR. Through a requirement analysis (see Chapter 3), we selected OrientDB as the most appropriate database solution and Gremlin as the language to retrieve data from the database.

We extended TESTAR with a module that implements the interface between TESTAR and the database. In the process of creating this module, we changed the build system of the TESTAR project from Apache Ant to Gradle (see Appendix B).

In Section 9.2 and 9.3 we presented the results of a case study. We looked at the usability and the performance impact. The graph database extension provides several benefits, such as increased flexibility on the information stored during a test and the possibility to perform off-line analyses on the data gathered. With the basic implementation used in this research, the execution time of test with 5000 samples increased by forty nine percent (see Table 9.12). Furthermore, the extension requires an extra twelve percent (see Table 9.15 and Table 9.16). The performance impact can be reduced choosing another design for the implementation of the database interface. We will now continue this section with a summery of the subjects discussed in the various sections.

Chapter 4 presented the design of the module `graphdb` and the way the existing code interacts with this module. We choose to store States, Widgets and Actions from the protocol execution flow. This answers the first part (How can we store the test results of TESTAR in a database) of Question Q1.

In Section 3.3 we established the requirements for the solution. Considering these requirements a database that is a pure graph database does not meet all our requirements. A multi-model database with the capabilities of key-value store and the capabilities of a graph database is is a better match to store the artefacts from TESTAR. OrientDB is a multi-model since it has the properties of a graph database (the way relations are modelled using vertices and edges), a key-value store (the way properties are added to vertices or edges) and a document database. This answers the second part (Is a graph database the most desirable type?) of Question Q1.

In Chapter 5, we explained how the data model of TESTAR can be extended. First, we showed how extra tags can be added to a type. Second, we showed how common properties can be grouped using a custom type. Finally, we explained the role of the method `toString` in the process of storing properties in the graph database. Using the facilities to add properties, custom types and implementing your own `toString` implementation, provides all options for the user the information required for the test scenarios. This answers Question Q2 (How we retrieve extra information from an application just by storing this extra information in the customized code of the protocol?).

In Section 9.2, we used the graph database to retrieve the information currently available in the log folder of TESTAR. The study showed that a lot of information current available can be retrieved from the database using Gremlin. However, not all data could be retrieved. An important short coming for the current implementation is the fact that the information which TESTAR stores in its internal model is not completely covered by the information stored in the database. This part of the model contains, among others information about the number of unexplored Actions and States. These numbers are important when we want to investigate test coverage. This answers Question Q3 (How does recording test results for a run with TESTAR in a database and the current way of storing test results compare?)

In Section 9.3 we investigated the performance impact as a result of storing test results in a graph database. As expected, storing data in the graph database does increase the execution time of a test run. For a test with 5000 steps (actions), the execution time increases from 54 minutes to 82 minutes. Most likely the impact on the execution time can be limited by offloading the database interaction into a separate thread. The required disk per step (action) is about 5.91 kilo bytes. The original TESTAR code requires 47,44 kilo bytes per action. This answers Question Q4 (What is the performance impact of using a graph database during a TESTAR run?).

In Section 3.4.2, we selected Gremlin as the tool to execute uniform queries. With uniform queries we mean queries that are independent from a specific manufacturer. Although other tools exist, We selected Gremlin as it is the only tool which is supported by OrientDB. During the case-study we found out that OrientDB version 2.2.x uses an old version of Gremlin with limited possibilities. The next major version of OrientDB (3.x) will provide support for the most recent version of Gremlin. Updating TESTAR is left as future work (see Section 10.1.1). This answers Question Q5 (How can we execute uniform queries on the data gathered by TESTAR?).

10.1.1. FUTURE WORK

During the evaluation of the usability of a graph database in the context of TESTAR and the impact of the graph database on the execution time and disk usages, we discovered a number of future improvements. These improvements are listed in this section.

- In our research we selected the core objects from the data model of TESTAR to be modelled in the database. In Section 9.2 we learned that the selected implementation does not allow us to retrieve all information which is available in the original output. Information like unexplored states and the exploration curve cannot be retrieved. This issue can be solved by storing information available in the `TestarEnvironment` into the database.
- Currently, the interface `GraphDBRepository` contains an API for storing data in the database as well as an API to query the database. These two concerns need to be split into separate interfaces.
- TESTAR does not distinguish different test runs in its output data. Dividing the data

in a folder for each test run would provide a better overview of which data belongs to which test run.

- In our research, we used Gremlin to query the underlying database. We used OrientDB version 2.2.x which only supports an old version of Gremlin. As of version 3.x of OrientDB, OrientDB will support the official Tinkerpop[1] release of Gremlin. This version has a much richer interface and is standardized. In a future research TESTAR needs to be updated to the new version of OrientDB. In order to use the new version of OrientDB version for the dependency in the Gradle build file needs to be updated. After this update the official gremlin version can be used. Information on how to run Gremlin in this scenario can be found at `https://orientdb.com/docs/3.0.x/tinkerpop3/OrientDB-TinkerPop3.html`.

- In the solution presented in our research the interaction with the graph database is performed on the same thread as the execution of the test. In a future project the interaction with the graph database shall be moved to a separate thread analogue to the mechanism used in the `TestSerializer`.

- Although the implementation provides the user the ability to add properties to a type and add custom types, there is no option to select the properties stored. A filter on the properties needs to be added to full fill this requirement. Filtering properties, which are not required for a test scenario, will have a positive effect on the required disk space.

- When the graph database is used, TESTAR will store all Actions, Widgets and States it encounters. It would be an improvement when the user can select the abstraction level allowing to only store States and there relation, States and Actions or other combinations. Storing information at an higher abstraction level will provide an opportunity to use the visualization options of OrientDB (see Section 9.2.1) since the performance of the visualization is poor when a large number of objects are presented. It will also benefit the overall performance of the extension.

[1] `http://tinkerpop.apache.org`

10.2. ACCESSIBILITY EVALUATION

In this section, we conclude on the research performed by Davy Kager. Davy investigated accessibility standards for desktop software and implemented accessibility evaluation in TESTAR using rules from the most suitable standard.

In Chapter 6 we formulated requirements to select a suitable accessibility standard, explored the available standards and selected WCAG 2.0. WCAG 2.0 was the only suitable standard because it is well-established, generic and applicable to desktop software through WCAG2ICT.
In Chapter 7 we defined oracles to perform accessibility evaluation with the accessibility rules from WCAG2ICT. A completely new type of oracle that uses graph database queries, called off-line oracles, made it possible to evaluate more accessibility rules on top of the ones defined with the traditional, on-the-fly oracles.
In Chapter 8 we presented the implementation in TESTAR, based on the oracles we had defined in the previous chapter.

In a case study that we presented in Section 9.4 we validated the evaluation report from TESTAR. A comparison between the results from TESTAR and our own, manually obtained results showed that TESTAR can complement but not replace manual evaluation. This is a finding that became apparent during the course of our research. WCAG2ICT contains accessibility rules that cannot be automated, so the need for expertise was immediately apparent. The case study confirmed that TESTAR adds real value to a manual accessibility evaluation.

We formulated four research questions for this sub-project (see Section 1.1.2). Chapter 6 answers Question Q6: What is a suitable accessibility standard to evaluate the accessibility of desktop software? In Section 6.1 we introduced the concept of accessibility standards and defined requirements for a standard suitable to apply to desktop software. Section 6.2 then explains why WCAG2ICT, based on WCAG 2.0, is the accessibility standard we used throughout the rest of the research. To narrow down what we mean by automated accessibility evaluation, Section 6.3 classified the method using TESTAR as guideline-based analysis.

Chapter 7 answers Question Q7: Which oracles can be defined for the selected accessibility standard? In Section 7.1 we described the two Windows accessibility APIs that TESTAR uses as the source of oracle information and chose UIA to define our oracles. By adding a graph database to TESTAR, it became possible for the first time to define oracles with oracle information that spans multiple states. Section 7.2 placed the traditional, on-the-fly oracles and the new, off-line oracles within the TESTAR execution flow. The oracle verdicts are reported using the evaluation report structure from Section 7.3. We defined on-the-fly and off-line oracles for the accessibility rules in WCAG2ICT. Section 7.4 lists these oracles and in Section 7.5 we reflected on these definitions.

Chapter 8 answers Question Q8: How can TESTAR be extended to support accessibility evaluation? In Section 8.2, 8.3 and 8.4 we presented our implementation of accessibility evaluation in TESTAR that consists of three main extensions. The respective sections dis-

cuss how we extended TESTAR with implementations for accessibility evaluation protocols, accessibility standards and evaluation results. In Section 8.5 we completed the presentation with the two utility classes that we added.

Section 9.4 answers Question Q9: How do experts benefit from our implementation of accessibility evaluation in TESTAR? Here we brought the work from the previous three chapters together in a case study. We concluded that TESTAR can support an expert by evaluating automatable accessibility rules defined as oracles, but that expertise continues to matter. TESTAR can only complement expert evaluation.

10.2.1. FUTURE WORK

Our case study showed that TESTAR is a promising automated analysis tool to support accessibility evaluation with minimal effort from the expert by inspecting the GUI for violations, but also that it needs refinement.

- In Section 6.1 we selected WCAG2ICT as the only desktop accessibility standard we found, which fortunately also satisfied all of our requirements. Research into desktop accessibility is a relatively unexplored topic compared to the large body of research on web accessibility. WCAG2ICT is intended as guidance to WCAG 2.0. More extensive accessibility evaluation could be performed with a standard tailored to the desktop.

- UIA is the general-purpose accessibility API that TESTAR uses on Windows. Section 7.1 pointed out that it does not expose colour information, so the contrast ratio between foreground and background colours cannot be calculated with TESTAR. There are more properties missing in TESTAR, such as text formatting information. Other APIs or techniques could be added to TESTAR to complement the accessibility APIs and the graph database.

- When reflecting on the accessibility oracles in Section 7.5 we already noted that there are not many WCAG2ICT oracles, especially off-line ones, because expertise matters and many success criteria cannot be automated or can only be partially automated. While this continues to be true, we expect that more oracles could be defined, either for WCAG2ICT or for another accessibility standard such as WCAG 2.0, especially when knowledge of the SUT can be exploited. Another option is to chain actions, for example to perform a right-click and, if a context menu appears, to then try the `Application` key that should open the same menu. The graph database can be used to compare the results from the right-click and the key press.

- Almost the entire implementation of accessibility evaluation in TESTAR is independent of specific technologies. The exception is the accessibility utility class discussed in Section 8.5. As TESTAR itself runs on other platforms besides Windows, a logical next step is to implement the utility class with accessibility APIs of other platforms. This may be complicated by the fact that platforms differ in their GUI and functionality.

- False positives and violations outside the SUT were a real problem during the case study. Part of the problem is the novelty of the accessibility implementation in TESTAR. An example is the bug in TESTAR that was uncovered via the unknown role oracle that we mentioned in Section 9.4.2. Evaluating more software with TESTAR could go a long way towards making the code more robust.

- After validating the evaluation report from TESTAR we already concluded in Section 9.4.3 that it could be made more readable and useful. TESTAR currently stores evaluation results in the graph database as simple strings and generates a static HTML report from this. The results could instead be stored in the graph database as vertices and exported in a machine-readable format, for example JSON. A web front-end could then be built to display a flexible, customizable report in the web browser using that exported data as its information source. Taking this a step further would even allow running queries straight from the report.

- Our research focused on desktop accessibility evaluation with TESTAR. Evaluating web accessibility with TESTAR does not seem very appealing at first glance, because TESTAR does not have access to the source code of a website. However, TESTAR has the big advantage that it can evaluate the accessibility of a website loaded in a browser and reported to assistive technologies through the accessibility API. This kind of evaluation could provide insight into the user experience of a website as viewed in combination with specific software. Our implementation of WCAG2ICT already lay the foundation for an implementation of WCAG 2.0.

A

CONFIGURING THE GRAPH DATABASE IN TESTAR

With the introduction of the graph database as storage in TESTAR, the configuration of TESTAR has been extended. This appendix describes the needed extension of the user manual [37]. The configuration can be performed through the settings dialogue (see Section A.1) or by directly editing the settings file (see Section A.2).

A.1. THE USER INTERFACE

When TESTAR is started and the GUI is enabled (`ShowVisualSettingsDialogOnStartup = true`), the graph database can be configured through the "GraphDB" tab in the settings dialogue. Figure A.1 shows this tab. Table A.1 shows the fields which can be configured.

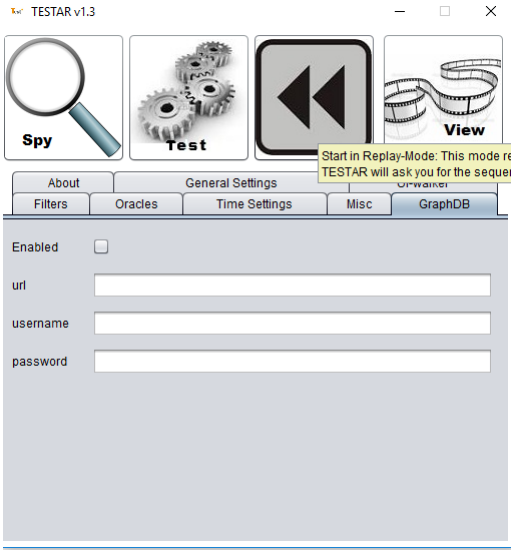


Figure A.1: The TESTAR settings dialogue for the graph database

Field	Description
Enabled	Information will be written to the graph database when this check-box is checked.
url	The Uniform Resource Locator (URL) has the following format: `<storage>:<location>/<database name>`. **storage** defines the type of storage used. Three types are supported: remote, memory and plocal (see `http://orientdb.com/docs/2.2.x/Storages.html` for more information). **location** provides the location of the database. This is either the host name of the machine where the database is running (for storage type remote) or the folder where the data is stored (for storage plocal). Finally, the **database name** is the name of the database where the data will be stored.
username	The user name used to interact with the database. For OrientDB this should be admin[1] by default
password	The password for the account used to interact with the database.

Table A.1: Description of the graph database fields in the TESTAR settings dialogue

A.2. THE SETTINGS FILE

The settings for the graph database can also be configured through the settings file. Table A.2 shows the variables and the mapping to their corresponding fields in the settings dialogue:

Variable	Field in the settings dialogue	Allowed values
GraphDBEnabled	Enabled	true or false
GraphDBURL	url	valid URL
GraphDBUser	username	text
GraphDBPassword	password	text

Table A.2: Description of the graph database fields in the TESTAR settings file

A.3. THE PREFERRED STORAGE TYPE

In Table A.2 storage is mentioned as one of the parts of the URL. OrientDB recognizes three types: remote, memory or plocal. The storage type remote requires that an OrientDB server is running on a location in the network. This is useful to offload the storage of the data to another machine. The storage type memory is not useful within TESTAR since the data will be lost as soon as the execution of TESTAR is finished. The storage type plocal is the most convenient option to use with TESTAR. It is this approach which is used in the performance evaluation (see Section 9.3).

[1] When the database is created with default settings a user admin will be created

B

THE GRADLE BUILD

The TESTAR project was originally built with Apache Ant[1]. Using XML files, a configuration was created that allowed the user to build the complete project from scratch. At the start of our research we noted that the build structure had its limitations, in particular in the presence of unit tests.

1. Unit tests cannot be added without changing the build script.
2. Due to the large amount of required "boiler-plate" code it is hard to detect the advanced parts of the build files.
3. There is a lot of duplication among the different Apache Ant files.

B.1. WHY GRADLE

Due to the limitations mentioned at the start of this appendix and to get a grip on the TESTAR code, we decided to introduce the Gradle[2] build system next to the Apache Ant build system that remained as backup. When enough confidence has been obtained by testing the Gradle build, the Apache Ant build can be removed.

As explained by Dockter et al. [12], Gradle is a tool that has the flexibility of Apache Ant and the convenience of Apache Maven[3]. Apache Ant is good at performing custom tasks like moving files while Apache Mavenis good at dependency management. Both features play an important role during the build of TESTAR, therefore we selected Gradle as build tool. In order to understand the impact from the introduction of Gradle we highlight several advantages (Section B.1.1) and disadvantages (Section B.1.2) of the new Gradle build system with respect to the original Apache Ant build.

[1] http://ant.apache.org
[2] gradle.org
[3] https://maven.apache.org

B.1.1. ADVANTAGES

- Gradle has a plugin system that provides some preconfigured tasks, for instance to build Java code or to measure code coverage.
- Using the dependency management system combined with the Apache Mavenrepository[4] allows us to remove third-party libraries from the code archive of the project and explicitly formalizes these dependencies.

B.1.2. DISADVANTAGES

- The Java plugin of Gradle expects a default folder structure. When the folder structure deviates from the default, the configuration needs to be customized. This can be quite cumbersome. Section B.4 explains which customization was performed.
- The dependency system requires an internet connection when the project is built the first time.

B.2. THE GRADLE WRAPPER

To allow developers of TESTAR to build the software without any tool installation up front, the TESTAR folder structure is extended with a Gradle wrapper. The wrapper consists of "gradlew" (a shell script for Linux and Mac), "gradlew.bat" (the equivalent batch file for Windows) and a Gradle folder that contains "gradle-wrapper.jar" and "gradle-wrapper.properties". The last file contains information about the required Gradle version for building the software. When the build is executed for the first time, the configured Gradle version is downloaded and the build is started with the proper tools. When a new version of Gradle should be used, only "gradle-wrapper.properties" need to be updated.

[4]The Apache Mavenrepository is an online archive (`https://mvnrepository.com`) where all major third-party libraries are stored.

B.3. STRUCTURE OF THE BUILD

The software archive of TESTAR consists of a number of modules which can be built separately. Every module performs a specific role in the application. Table B.1 presents these modules and their role in TESTAR.

Module	Description
core	Implements the core model of TESTAR. Types like `Widget`, `State` and `Action` are defined in this module.
graph	Implements the internal graph model used within TESTAR. This model contains the runtime state of TESTAR.
graphdb	Provides an interface to a graph database. It allows TESTAR to store data in a graph database.
linux	Provides a dummy implementation that could be extended to allow TESTAR to interact with the accessibility API of the Linux operating system.
native	Provides an abstraction layer on top of the accessibility API of the operating system.
testar	The main module that provides the entry point to start TESTAR and contains test protocols. The build of this module performs the integration of all the modules into a single application.
windows	Implements the interaction between TESTAR and the accessibility API of the Windows operating system.

Table B.1: Overview of the modules within the code archive of TESTAR

The build configuration consists of the following artefacts in the root of the TESTAR project:

settings.gradle Contains one `include` statement with a comma-separated list of the modules that make up the project.

build.gradle The top level build file which loads the "eclipse" plugin[5] and defines the default configuration which holds for all modules in the sub-projects section (see Section B.4).

Each module (see Table B.1) of the project contains a "build.gradle" file. This file extends the build configuration present in the sub-projects section of the "build.gradle" file at root level. A model of this relation is presented in Figure B.1.

Most modules only contain information about the dependencies of the component. The "build.gradle" from the core module is empty since core is a standard Java library without any requirements for third-party plugins. The windows module contains two custom tasks, these are required to invoke the build of the Microsoft Windows API required for the JNI layer.

[5]The "eclipse" plugin adds functionality to allow the Gradle build to be imported into Eclipse (see https://docs.gradle.org/current/userguide/eclipse_plugin.html).

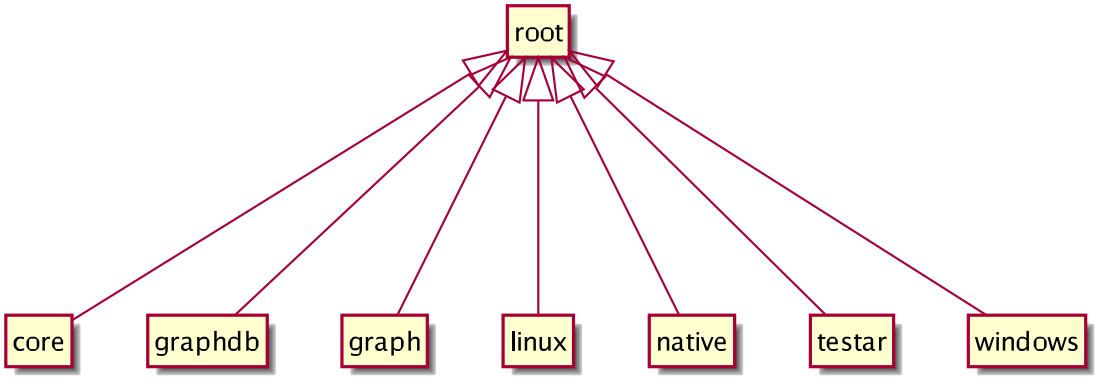


Figure B.1: The relation between the different “build.gradle” files of the TESTAR Gradle project

B.4. SUB-PROJECTS

The following code snippet of the “build.gradle” file at the root of the project shows the contents of the sub-projects definition.

Listing B.1: Common settings for the Gradle build of each module

```
subprojects {
 apply plugin: 'java'
 apply plugin: 'jacoco'
 sourceSets {
  main {
   java {
    srcDirs = ['src']
   }
  }
  test {
   java {
    srcDirs = ['test']
   }
  }
 }
 repositories {
  mavenCentral()
 }
 dependencies {
  testCompile group:'junit', name:'junit', version:'4.12'
 }
 buildDir = new File('./target')
}
```

The `subprojects` definition is applied to every module included in the `include` statement from the “settings.gradle” file. The definition starts by applying the “java” and “jacoco” plugins. The “java” plugin provides all the build tasks required to build Java code. For more information see https://docs.gradle.org/current/userguide/java_plugin.html. The “jacoco” plugin provides tasks that support code coverage measurements. For more information see https://docs.gradle.org/current/userguide/jacoco_plugin.html.

The `sourceSets` part of the `subprojects` definition overrules the default path where the “java” plugin looks for source code. This is required since the folder structure of the TESTAR project does not follow the default project structure. Instead of the default location

for Gradle, “src/main/java”, TESTAR uses “src”. This is what we meant with cumbersome in the disadvantages mentioned in Section B.1.

The `repositories` part tells Gradle that the Apache Mavencentral repository is used to resolve external dependencies. The dependency on the JUnit test framework holds for all modules. Finally, the output folder for each module is set to `<module_root>/target` instead of the default `<module_root>/build` folder.

B.5. BUILDING WITH GRADLE

B.5.1. HOW TO USE GRADLE

TESTAR can be built with Gradle using the command line (see Section B.5.2) or an Integrated Development Environment (IDE) that supports Gradle. Most modern IDEs support Gradle. For instance, Eclipse[6], Netbeans[7], IntelliJ IDEA[8] or Visual Studio Code[9]. The software project can be opened as Gradle project in IntelliJ IDEA. The Readme for the software archive (see `https://github.com/TESTARtool/TESTAR_dev` provides information on how to open the Gradle build in Eclipse. Instructions on how to load the project in Netbeans or Visual Studio Code are left to the reader. We continue the discussion with an explanation of the Gradle wrapper.

B.5.2. IMPORTANT BUILD COMMANDS FOR THE COMMAND LINE

Table B.2 presents an overview of the important Gradle commands which can be used to build and verify the TESTAR project. Before starting a build, read the "Readme.md" file at the root of the project. This document provides all information required to set up the build environment.

Command	Description
gradlew distZip	This command is only available at the root of the project. It builds and tests the TESTAR project. It also packages all the required artefacts in a zip file with the required structure to unzip and run TESTAR.
gradlew test	Builds the software (classes and jar files) and runs all unit tests. This command does not generate a distributable package.
gradlew jacocoTestReport	Runs the tests and generates a code coverage report.

Table B.2: Important Gradle commands

The list of tasks only reflects the most import ones. Use `gradlew tasks` to display a complete list of available tasks.

[6] `http://www.eclipse.org`
[7] `https://netbeans.org`
[8] `https://www.jetbrains.com/idea`
[9] `https://code.visualstudio.com`

B.6. FUTURE IMPROVEMENTS

To further improve the build system in the future, we suggest the following changes:

- Remove the Apache Ant build system.
- Put the windows module into a separate project with its own life cycle. Make the build result (publicly) available. This allows developers on Mac[10]or Linux[10] to build the TESTAR project on their own machine.
- Remove the binaries from third-party libraries[11] and replace them with a reference to the Apache Mavenrepository.
- Minimize the use of third-party libraries and use current versions whenever possible.

[10]The only reason TESTAR cannot be built on Mac or Linux is the Windows native code. A build for Mac or Linux has only added value when a module is created to access the accessibility API of this operating systems.

[11]The following libraries are currently part of the TESTAR archive: jiprolog-4.1.3.1.jar, jgrapht-ext-1.0.1-uber.jar, libatspi.jar, junit-jupiter-api-5.0.0-M3.jar, com.alexmerz.graphviz.jar, jsyntaxpane-1.1.5.jar and JNativeHook.jar.

C

EXPORT AND IMPORT OF A ORIENTDB DATABASE

There are scenarios when a OrientDB database needs to be moved to another environment. One of these scenarios is the case where we want to use the visual interface available in server mode. As explained in Appendix A, OrientDB supports three storage types: plocal, remote and memory. Only plocal and remote can be used for scenarios where we want to perform off-line analysis. When plocal is used, the data needs to be moved to a remote OrientDB instance in order to use the visual graph inspection feature of OrientDB. This appendix describes the process. Both import and export are performed using the tool "console.bat" which is part of the OrientDB installation folder. In this appendix we assume OrientDB server is running and a terminal is opened in the folder which contains a OrientDB named "demo".

C.1. EXPORT THE DATABASE

Perform the following procedure to export the demo database:

1. Start "console.bat"[1].
2. At the prompt of the console, run the following command:
   `CONNECT "plocal:./demo"admin admin` to connect to the database.
3. At the prompt of the console, run the following command:
   `EXPORT DATABASE demo.export` to export the database.
4. At the prompt of the console, run the following command:
   `quit` to exit the console.

[1] On Linux and Mac this command is "orientdb-console"

See https://orientdb.com/docs/last/Console-Command-Export.html for more information on the export command.

C.2. IMPORT THE DATABASE

Perform the following procedure to import the demo database:

1. Start “console.bat”.
2. Create the database:
   `CREATE DATABASE remote:/localhost/demo root root password plocal graph`[2].
3. At the prompt of the console, run the following command:
   `CONNECT "remoet:/localhost/demo"admin admin` to connect to the database.
4. At the prompt of the console, run the following command:
   `IMPORT DATABASE demo.export.gz` to import the database.
5. At the prompt of the console, run the following command:
   `quit` to exit the console.

[2]The root password used during the installation of OrientDB

D

API OF THE MODULE GRAPHDB

Listing D.1: The GraphDB interface (first part)

```
public interface GraphDBRepository {

 /**
 * Store State in Graph database.
 *
 * @param state State of the SUT for this step.
 * @param isInitial indicate if the state is initial.
 */
 void addState(final State state, final boolean isInitial);

 /**
 * Add Action on a widget to the graph database as Edge
 *
 * @param action   The performed action
 * @param toStateID  ConcreteID of the new State
 */
 void addAction( final Action action, final String toStateID);


 /**
 * Add Action on a State to the graph database as Edge
 *
 * @param stateId id of the state on which the action is performed.
 * @param action the action.
 * @param toStateID the resulting state
 */
 void addActionOnState(final String stateId, final Action action, final String toStateID);
```

Listing D.2: The GraphDB interface (second part)

```
 /**
 * Add a widget to the the graph database as Wiget.
 * @param stateID State to which the widget belongs
 * @param w The widget object
 */
 void addWidget(final String stateID, Widget w);

 /**
 * Store a custom type in the graph database.
 * @param action the actionto which the custom type relates.
 * @param relation The name of the relation
```

```
     * @param instance the custom object.
     */
    void addCustomType(final Action action, final String relation, final CustomType instance);

    /**
     * Store a custom type in the graph database.
     * @param state the ID of the artifact to which the custom type relates.
     * @param relation The name of the relation
     * @param instance the custom object.
     */
    void addCustomType(final State state, final String relation, final CustomType instance);

    /**
     * Store a custom type in the graph database.
     * @param widget the ID of the artifact to which the custom type relates.
     * @param relation The name of the relation
     * @param instance the custom object.
     */
    void addCustomType(final Widget widget, final String relation, final CustomType instance);

    /**
     * Get all objects from a pipe specified by a Gremlin-Groovy expression
     * @param gremlin The Gremlin-Groovy expression.
     * @param start The starting point for the expression.
     * @return A list of all objects in the pipe.
     */
    List<Object> getObjectsFromGremlinPipe(String gremlin, GremlinStart start);
}
```

E

USING THE GREMLIN CONSOLE

One way to query the graph database is through the use of the Gremlin console. This appendix explains how to obtain (Section E.1) and use (Section E.2) the console. The instruction applies to OrientDB version 2.x.

E.1. GETTING THE PREREQUISITES

In order to use Gremlin on a local machine with the OrientDB database, it is required to install the community edition of OrientDB on that machine. The proper zip file with the installation components can be found at `https://orientdb.com/download-previous`.

When the installation is done, create an environment variable `ORIENTDB_HOME` that points to the installation root folder of the OrientDB installation. The system is now ready to query an OrientDB database.

E.2. RUNNING THE CONSOLE

The following procedure explains how the Gremlin console can be used to open a database created by TESTAR. The example expects that the database was created in "
tmp" and is called "demo".

1. Change directory: `cd \tmp`
2. Start the Gremlin console: `%ORIENTDB_HOME%\bin\gremlin.bat`
3. Open the Graph: `G = new OrientGraph(''plocal:demo'',''admin'',''admin'')`
4. Now the database is loaded.

F

Settings Used for the Performance Measurements on TESTAR

```
TypingTextsForExecutedAction = 10
SUTConnectorValue = C:\\Program Files (x86)\\VideoLAN\\VLC\\vlc.exe
Discount = 0.95
GraphDBEnabled = true
ProtocolClass = desktop\_generic\_graphdb/Protocol\_desktop\_generic\
   _graphdb
TempDir = ./output/temp
ClickFilter = .*[sS]istema.*|.*[sS]ystem.*|.*[cC]errar.*|.*[cC]lose
   .*|.*[sS]alir.*|.*[eE]xit.*|.*[mM]inimizar.*|.*[mM]inimi[zs]e.*|.*[gG
   ]uardar.*|.*[sS]ave.*|.*[iI]mprimir.*|.*[pP]rint.*
MaxTime = 3.1536E7
Sequences = 50
GraphDBUrl = plocal:output/run\_50x100
StopGenerationOnFault = true
PrologActivated = false
GraphDBUser = admin
ExecuteActions = true
DrawWidgetTree = false
UseRecordedActionDurationAndWaitTimeDuringReplay = false
SequenceLength = 100
ForceForeground = true
Mode = Generate
AlgorithmFormsFilling = false
FaultThreshold = 1.0E-9
DrawWidgetInfo = false
MaxReward = 9999999.0
LogLevel = 1
VisualizeSelectedAction = true
AccessBridgeEnabled = false
```

```
Delete =
MyClassPath = ./settings
ReplayRetryTime = 30.0
ProcessesToKillDuringTest =
ShowSettingsAfterTest = true
ForceToSequenceLength = true
OutputDir = ./output
UnattendedTests = false
TestGenerator = random
StartupTime = 2.0
ActionDuration = 0.0
NonReactingUIThreshold = 100
GraphsActivated = true
GraphResuming = true
GraphDBPassword = admin
CopyFromTo =
TimeToFreeze = 30.0
StateScreenshotSimilarityThreshold = 1.4E-45
ShowVisualSettingsDialogOnStartup = true
SUTProcesses =
TimeToWaitAfterAction = 0.1
DrawWidgetUnderCursor = true
SUTConnector = COMMAND\_LINE
SuspiciousTitles = .*[eE]rror.*|.*[eE]xcep[ct]i[o?]n.*
PathToReplaySequence = E:\\testar\\bin\\output\\sequences\
    _unexpectedclose\\sequence37
VisualizeActions = false
ExplorationSampleInterval = 1
OnlySaveFaultySequences = false
OfflineGraphConversion = true
```

G

SETTINGS USED FOR THE ACCESSIBILITY EVALUATIONS WITH TESTAR

TypingTextsForExecutedAction<Integer> : 10
SUTConnectorValue<String> : C:\Program Files (x86)\VideoLAN\VLC\vlc.exe
Discount<Double> : 0.95
GraphDBEnabled<Boolean> : true
ProtocolClass<String> : accessibility_wcag2ict/Protocol_accessibility_wcag2ict
TempDir<String> : ./output/temp
ClickFilter<String> : .*[sS]ystem.*|.*[cC]lose.*|.*[eE]xit.*|.*[mM]inimi[zs]e.*|.*[sS]ave.*|.*[pP]rint.*|.*[fF]ile.*|.*[dD]elete.*
MaxTime<Double> : 3.1536E7
Sequences<Integer> : 1
GraphDBUrl<String> : plocal:./output/logs/db
StopGenerationOnFault<Boolean> : true
PrologActivated<Boolean> : false
GraphDBUser<String> : admin
ExecuteActions<Boolean> : true
DrawWidgetTree<Boolean> : false
UseRecordedActionDurationAndWaitTimeDuringReplay<Boolean> : false
SequenceLength<Integer> : 50
ForceForeground<Boolean> : true
Mode<Modes> : Generate
AlgorithmFormsFilling<Boolean> : false
FaultThreshold<Double> : 1.01
DrawWidgetInfo<Boolean> : false
MaxReward<Double> : 9999999.0
LogLevel<Integer> : 2
VisualizeSelectedAction<Boolean> : false
AccessBridgeEnabled<Boolean> : false
Delete<List> : []
MyClassPath<List> : [./settings]
ReplayRetryTime<Double> : 30.0
ProcessesToKillDuringTest<String> :
ShowSettingsAfterTest<Boolean> : true
ForceToSequenceLength<Boolean> : true
OutputDir<String> : ./output
UnattendedTests<Boolean> : false
TestGenerator<String> : random
StartupTime<Double> : 5.0
ActionDuration<Double> : 0.1
NonReactingUIThreshold<Integer> : 100
GraphsActivated<Boolean> : true
GraphResuming<Boolean> : true
GraphDBPassword<String> : admin
CopyFromTo<List> : []
TimeToFreeze<Double> : 30.0
StateScreenshotSimilarityThreshold<Float> : 1.4E−45

ShowVisualSettingsDialogOnStartup<Boolean> : true
SUTProcesses<String> :
TimeToWaitAfterAction<Double> : 0.1
DrawWidgetUnderCursor<Boolean> : false
SUTConnector<String> : COMMAND_LINE
SuspiciousTitles<String> : .*[eE]rror.*|.*[eE]xception.*
PathToReplaySequence<String> : ./output/temp
VisualizeActions<Boolean> : false
ExplorationSampleInterval<Integer> : 1
OnlySaveFaultySequences<Boolean> : false
OfflineGraphConversion<Boolean> : true

Acronyms

API Application Programming Interface. 5, 18, 22, 25, 41, 42, 43, 45, 46, 47, 51, 58, 62, 63, 64, 65, 68, 77, 78, 79, 90, 92, 96, 110, 114, 125, 127, 128, 129, 135, 139

CR Capture Replay. 1, 5, 11, 12

CSV Comma-Separated Values. 11, 14

DBMS Database Management System. 16

GUI Graphical User Interface. xiii, 5, 6, 8, 9, 11, 14, 20, 43, 47, 64, 110, 112, 114, 128, 131

HTML HyperText Markup Language. 12, 14, 77, 78, 114, 129

IDE Integrated Development Environment. 138

JAB Java Access Bridge. 46, 62

JMH Java Micro benchmark Harness. 90, 91, 94

JSON JavaScript Object Notation. 84, 129

NOP No Operation. 64

NVDA NonVisual Desktop Access. 110, 113, 114

OCR Optical Character Recognition. 45

PDF Portable Document Format. 38, 40, 41

PDF/UA PDF/Universal Accessibility. 38

RFT Rational Functional Tester. 11, 12, 14

SQL Structured Query Language. 18, 87

SUT System Under Test. ix, 6, 8, 9, 14, 16, 21, 22, 27, 45, 46, 48, 55, 57, 58, 62, 64, 65, 71, 81, 82, 89, 96, 108, 110, 117, 118, 119, 120, 121, 122, 128

UIA UI Automation. 46, 53, 55, 58, 61, 62, 64, 68, 77, 79, 127, 128

URL Uniform Resource Locator. 131, 132

VGT Visual GUI Test. 1, 5, 11, 12

W3C World Wide Web Consortium. 39, 40

WAI Web Accessibility Initiative. 40

WAQM Web Accessibility Quantitative Metric. 49

WCAG Web Content Accessibility Guidelines. ix, x, xiii, 38, 39, 40, 41, 42, 43, 49, 51, 52, 54, 56, 58, 59, 61, 67, 73, 75, 77, 109, 127, 128, 129

WCAG-EM Website Accessibility Conformance Evaluation Methodology. 109, 111

BIBLIOGRAPHY

[1] Pekka Aho, Matias Suarez, Teemu Kanstren and Atif M. Memon. 'Murphy tools: Utilizing extracted GUI models for industrial software testing'. In: *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014* (2014), pp. 343–348. ISSN: 2159-4848. DOI: 10.1109/ICSTW.2014.39.

[2] Emil Alégroth, Robert Feldt and Lisa Ryrholm. 'Visual GUI testing in practice: challenges, problemsand limitations'. In: *Empirical Software Engineering* 20.3 (2015), pp. 694–744. ISSN: 15737616. DOI: https://doi.org/10.1007/s10664-013-9293-5.

[3] Emil Alégroth, Zebao Gao, Rafael Oliveira and Atif Memon. 'Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study'. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings* (2015). ISSN: 2159-4848. DOI: 10.1109/ICST.2015.7102584.

[4] Francisco Almenar, Anna I Esparcia-Alcázar, Mirella Martínez and Urko Rueda. 'Automated Testing of Web Applications with TESTAR'. In: *International Symposium on Search Based Software Engineering*. Springer. 2016, pp. 218–223.

[5] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter and Domagoj Vrgoc. 'Foundations of Modern Query Languages for Graph Databases'. In: V (2016). ISSN: 15577341. DOI: 10.1145/3104031. arXiv: 1610.06264. URL: http://arxiv.org/abs/1610.06264.

[6] S Bauersfeld and Tanja EJ Vos. 'GUITest: a Java Library for Fully Automated GUI Robustness Testing'. In: *EEE/ACM International Conference on Automated Software Engineering* (2012), pp. 330–333. DOI: 10.1145/2351676.2351739. URL: http://dl.acm.org/citation.cfm?id=2351739.

[7] Sebastian Bauersfeld, Tanja EJ Vos, Nelly Condori-Fernández, Alessandra Bagnato and Etienne Brosse. 'Evaluating the TESTAR Tool in an Industrial Case Study'. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, 4:1–4:9. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652588. URL: http://doi.acm.org/10.1145/2652524.2652588.

[8] Giorgio Brajnik, Yeliz Yesilada and Simon Harper. 'Testability and Validity of WCAG 2.0: the Expertise Effect'. In: *Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM. 2010, pp. 43–50.

[9] Giorgio Brajnik, Yeliz Yesilada and Simon Harper. 'The Expertise Effect on Web Accessibility Evaluation Methods'. In: *Human–Computer Interaction* 26.3 (2011), pp. 246–283.

[10] Ben Caldwell, Michael Cooper, Loretta Guarino Reid and Gregg Vanderheiden. 'Web Content Accessibility Guidelines (WCAG) 2.0'. In: (2008). URL: http://www.w3.org/TR/2008/REC-WCAG20-20081211/.

[11] CEN, CENELEC and ETSI. *Accessibility Requirements Suitable for Public Procurement of ICT Products and Services in Europe*. Standard EN 301 549. ETSI, 2015.

[12] Hans Dockter and Adam Murdoch. *Gradle userguide*. 2017. URL: https://docs.gradle.org/4.1/userguide/userguide.html (visited on 03/09/2017).

[13] Anna I Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda and Tanja EJ Vos. 'Q-learning Strategies for Action Selection in the TESTAR Automated Testing Tool'. In: *6th International Conference on Metaheuristics and Nature Inspired Computing*. 2016, pp. 130–137.

[14] Anna I Esparcia-Alcázar, Francisco Almenar, Urko Rueda and Tanja EJ Vos. 'Evolving Rules for Action Selection in Automated Testing via Genetic Programming – A First Approach'. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2017, pp. 82–95.

[15] EyeAutomate.com. *Eye Automate manual*. 2017. URL: http://eyeautomate.com/resources/EyeAutomateManual.html#h.901r5z2ib3t (visited on 08/12/2017).

[16] André P Freire, Renata PM Fortes, Marcelo AS Turine and Debora Paiva. 'An Evaluation of Web Accessibility Metrics Based on Their Attributes'. In: *Proceedings of the 26th Annual ACM International Conference on Design of Communication*. ACM. 2008, pp. 73–80.

[17] Andre P Freire, Christopher Power, Helen Petrie, Eduardo H Tanaka, Heloisa V Rocha and Renata PM Fortes. 'Web Accessibility Metrics: Effects of Different Computational Approaches'. In: *International Conference on Universal Access in Human-Computer Interaction*. Springer. 2009, pp. 664–673.

[18] Erich. Gamma. *Design patterns: elements of reusable object-oriented software*. Reading, Mass., 1995. URL: file://catalog.hathitrust.org/Record/002907294%20https://hdl.handle.net/2027/mdp.39015047433399.

[19] Mouna Hammoudi, Gregg Rothermel and Paolo Tonella. 'Why do Record/Replay Tests of Web Applications Break?' In: *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016* (2016), pp. 180–190. DOI: 10.1109/ICST.2016.16.

[20] IBM. *Functional Test Logs*. 2017. URL: https://www.ibm.com/support/knowledgecenter/en/SSJMXE_8.0.0/com.ibm.rational.test.ft.doc/topics/AboutViewingResultsintheLo
html (visited on 15/12/2017).

[21] ISO and IEC. *Information Technology – W3C Web Content Accessibility Guidelines (WCAG) 2.0*. Standard 40500:2012. ISO, 2012.

[22] Melody Y Ivory and Marti A Hearst. 'The State of the Art in Automating Usability Evaluation of User Interfaces'. In: *ACM Computing Surveys* 33.4 (2001), pp. 470–516.

[23] Peter Korn, Loïc Martínez Normand, Mike Pluke, Andi Snow-Weaver and Gregg Vanderheiden. 'Guidance on Applying WCAG 2.0 to Non-web Information and Communications Technologies (WCAG2ICT)'. In: (2013). Editors' Draft, work in progress. URL: http://www.w3.org/WAI/GL/2013/WD-wcag2ict-20130905/.

[24] Mirella Martínez, Anna I Esparcia-Alcázar, Urko Rueda, Tanja EJ Vos and Carlos Ortega. 'Automated Localisation Testing in Industry with Test*'. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2016, pp. 241–248.

[25] Microsoft. *Office Accessibility Center – Resources for People with Disabilities*. URL: https://support.office.com/en-us/article/Office-Accessibility-Center-Resources-for-people-with-disabilities-ecab0fcf-d143-4fe8-a2ff-6cd596bddc6d (visited on 22/01/2018).

[26] Microsoft. *UI Automation*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ee684009.aspx (visited on 25/02/2018).

[27] Grzegorz Molicki. *Testing your code performance with JMH*. 2017. URL: https://blog.goyello.com/2017/06/19/testing-code-performance-jmh-tool/ (visited on 20/10/2017).

[28] Oracle. *Javadoc class Object*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString() (visited on 11/02/2018).

[29] OrientDB. *plocal storage internals*. 2018. URL: https://orientdb.com/docs/2.2/Paginated-Local-Storage.html (visited on 04/01/2018).

[30] *OrientDB console Tutorial*. 2018. URL: http://orientdb.com/docs/last/Tutorial-Run-the-console.html (visited on 04/01/2018).

[31] Hans Persson, Henrik Åhman, Alexander Arvei Yngling and Jan Gulliksen. 'Universal Design, Inclusive Design, Accessible Design, Design for All: Different Concepts—One Goal? On the Concept of Accessibility—Historical, Methodological and Philosophical Aspects'. In: *Universal Access in the Information Society* 14.4 (2015), pp. 505–526. DOI: 10.1007/s10209-014-0358-z. URL: https://doi.org/10.1007/s10209-014-0358-z.

[32] W3C WAI Research and Development Working Group (RDWG). 'Research Report on Web Accessibility Metrics'. In: *W3C WAI Symposium on Website Accessibility Metrics*. Ed. by Markel Vigo, Giorgio Brajnik and Joshue O Connor. First Public Working Draft. W3C WAI Research and Development Working Group (RDWG) Notes. W3C Web Accessibility Initiative (WAI), 2012. URL: http://www.w3.org/TR/2012/WD-accessibility-metrics-report-20120830/.

[33] Marko a. Rodriguez. 'The Gremlin Graph Traversal Machine and Language'. In: *Proc. 15th Symposium on Database Programming Languages* (2015), pp. 1–10. DOI: 10.1145/2815072.2815073. arXiv: 1508.03843. URL: http://arxiv.org/abs/1508.03843%5Cnhttp://dx.doi.org/10.1145/2815072.2815073.

[34] Urko Rueda, Tanja EJ Vos, Francisco Almenar, Mirella Martínez and Anna I Esparcia-Alcázar. 'TESTAR: from Academic Prototype towards an Industry-ready Tool for Automated Testing at the User Interface Level'. In: *Jornadas de Ingeniería de Software y Bases de Datos (JISBD)* (2015), pp. 236–245.

[35] Solid-it. *DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems*. 2017. URL: https://db-engines.com/en/ (visited on 01/07/2017).

[36] TESTAR.org. *About – Test**. URL: http://webtestar.dsic.upv.es/index.php/about/ (visited on 14/04/2017).

[37] Testar.org. *TESTAR User Manual*. Tech. rep. September. 2017, p. 58.

[38] Eric Velleman and Shadi Abou-Zahra. 'Website Accessibility Conformance Evaluation Methodology (WCAG-EM) 1.0'. In: (2014). Working Group Note. URL: http://www.w3.org/TR/2014/NOTE-WCAG-EM-20140710/.

[39] Markel Vigo, Myriam Arrue, Giorgio Brajnik, Raffaella Lomuscio and Julio Abascal. 'Quantitative Metrics for Measuring Web Accessibility'. In: *Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility*. ACM. 2007, pp. 99–107.

[40] Markel Vigo and Giorgio Brajnik. 'Automatic Web Accessibility Metrics: Where We Are and Where We Can Go'. In: *Interacting with Computers* 23.2 (2011), pp. 137–155. DOI: 10.1016/j.intcom.2011.01.001.

[41] Markel Vigo, Giorgio Brajnik, Myriam Arrue and Julio Abascal. 'Tool Independence for the Web Accessibility Quantitative Metric'. In: *Disability and Rehabilitation: Assistive Technology* 4.4 (2009), pp. 248–263.

[42] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld and Joachim Wegener. 'Testar: Tool Support for Test Automation at the User Interface Level'. In: *International Journal of Information System Modeling and Design (IJISMD)* 6.3 (2015), pp. 46–83. ISSN: 1947-8186. DOI: 10.4018/IJISMD.2015070103. URL: http://dx.doi.org/10.4018/IJISMD.2015070103.

[43] Tanja EJ Vos, Urko Rueda Molina and Wishnu Prasetya. *Automated Testing at the User Interface Level*. ICT open, poster Dutch ICT-research. 2016.

[44] Qing Xie and Atif M Memon. 'Designing and Comparing Automated Test Oracles for GUI-based Software Applications'. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16.1 (2007). DOI: 10.1145/1189748.1189752. URL: http://doi.acm.org/10.1145/1189748.1189752.