MASTER'S THESIS

Intergrating TESTAR in a DevOps environment

Slomp, A.

Award date:
2021

INTEGRATING TESTAR IN A DEVOPS ENVIRONMENT

by

Arend Slomp

Student number:		
Course code:	IM9906	
Thesis committee:	dr. P. Aho,	Open University
	prof. dr. T. Vos,	Open University

Open Universiteit
de beste! www.ou.nl

This page is intentionally left blank

Name	Arend Slomp
Student number	
Title	Integrating TESTAR in a DevOps environment
Date of presentation	July 9th, 2021 9:00
Degree programme	Open University of the Netherlands, Faculty of Science, Master's Programme in Software Engineering
Graduation committee	prof. dr. T. Vos (chair) dr. P. Aho (primary supervisor) prof. dr. T. Vos (secondary supervisor)
Course code	IM9906

ACKNOWLEDGEMENTS

I want to thank my supervisors Pekka and Tanja for supporting the complete graduation process. It was a very nice time working on the graduation where a lot of new knowledge is acquired. The weekly meetings with Pekka, Fernando and optionally Tanja where very useful.

Furthermore I would like to thank my girlfriend for the support and the time given to complete the paper (even during our upcoming marriage and the garden becoming a complete mess)

Next I have to thank my company for the time being given off, even when it was almost not possible or on a very short notice.

Last but not least I want to thank everyone else who supported me during the writing of this thesis.

CONTENTS

1. SUMMARY

This thesis is about how TESTAR can be integrated in a Continuous Integration pipeline. TESTAR runs at this moment on a single node. For this reason, a test runs possibly for a long time before TESTAR finishes. In DevOps the software should be tested as soon as it is released. In the case of this company every night a release is done to the test environment (and should be tested as well). The research had as a goal to make TESTAR fit to run on a Kubernetes cluster where there is made use of Docker images. During this thesis a Docker image is created for TESTAR so it can be run from a Kubenetes deployment. The research had 3 questions: 1. "How can TESTAR perform all the tests on the SUT in parallel on a Kubernetes cluster", 2. “How can TESTAR be integrated in a CI/CD pipeline with the specific SUT” and a final integration question “How much faster is TESTAR when running on multiple pods when compared to a single instance?” It was an experimental research and the results will be presented.

2. INTRODUCTION

The research in this thesis will be about TESTAR applied in an industrial environment. TESTAR is a tool that is able to test software without pre-programmed scripts. Testing is relevant to the society due to the fact that it finds failures, so that they can be fixed before the public release of the software. In the industry, testing takes a lot of time and manual effort. The company that is used as a case study in this research has invested a lot of time and effort in setting up a good test environment. The issue with the current testing tools is that even minimal changes in the user interface (UI) will render the current scripts useless, since they are failing every time a change is made to the system under test (SUT). Repairing these failed tests is very time consuming. That is why the company wants to research a scriptless approach like TESTAR.

Several changes have to be made to TESTAR to be able to run in the testing process of the case study company.

First, we need to make it as fast as possible such that the company can keep up with short development cycles. At this moment, TESTAR can be run in parallel, however doing so does not take in account actions performed by other instances. With smarter action selection it is possible to let the nodes cooperate together and test the SUT as fast as possible. To be able to run TESTAR in parallel, the author chooses Docker images in combination with Kubernetes. Kubernetes is especially interesting due to the fact that it scales over a lot of nodes by just a few clicks or commands in the management interface of Kubernetes. Other advantages of running TESTAR from a Docker image is the lower amount of hardware resources needed for the specific instances. Details about Kubernetes are given in the background chapter of this thesis.

Second, TESTAR needs to be integrated into the CI/CD environment of the SUT and to be able to test the SUT. The SUT contains a lot of forms and TESTAR selects actions and fills text fields in a random way. By filling text fields in a web form randomly, TESTAR can take a long time to fill a form, if it even succeeds. The SUT, selected for testing in this study, has a registration process that contains a lot of forms that need to be filled correctly to be able to continue to the next steps. Hence, we need to extend TESTAR in such a way that it can fill a complete form as a single action (with potential values that make sense for the SUT) .

The company in which the research will be conducted has several test scripts written

to be executed after each automated deployment. TESTAR will be run in parallel next to this pipeline to do its own scriptless testing. The tooling used for automated deployment is Azure DevOps and is explained in 3 of this thesis.

TESTAR will be used to perform the UI tests, similar to how the current test automation works at this moment in the company. The most important part that cannot be tested by TESTAR are the functional tests where business rules and manual workflows are important. In the SUT of the company there exist several processes that require data input in fields. This data is business context specific and cannot be randomly guessed. TESTAR is not able to fill these fields with data that makes sense to the SUT. At this moment, such tests are being performed by the existing test tooling and it will not be expected from TESTAR that TESTAR can handle such contextual data.

Since the topic is about testing the SUT as fast as possible by doing the tests in parallel and visiting the paths in such a way that always the shortest path to an unvisited action is chosen, some modifications have to be made to TESTAR. The most important are:

- TESTAR has to be able to run containerized and be scalable so it can perform enough tests within a limited time. Running it containerized will provide the ability to run the TESTAR on a Kubernetes cluster. Running TESTAR on multiple nodes in parallel cooperating with each other will also be called distributed GUI testing. TESTAR itself is GUI testing, distributing it over many instances within Kubernetes we call it distributed GUI testing. The maximum time which is allowed to return the test results is two to four hours, depending on the planning of the release to the test environment. This limit is a business requirement dictated by the case study and not a technical requirement. As part of the parallelized testing it must as well be possible to report the bugs back in a central test system.

- TESTAR needs to be integrated in a build and release pipeline of the company providing the case study, so it runs at the deployment time and test results are available after the deployment.

- To deal with the specific SUT, TESTAR has to be able to fill in forms. Normally, TESTAR is used in an environment where it only performs all the possible actions. The system under testing (SUT) of the case study has a lot of forms to fill in and testing has to be performed on all those forms.

- TESTAR must be able to report the bugs it found back to the deployment tool used in the case study.

From the above goals the following research questions are taken:

- "RQ1 How can TESTAR perform all the tests on the SUT in parallel on a Kubernetes cluster"

- "RQ2 How can TESTAR be integrated in a CI/CD pipeline with the specific SUT?"

- "RQ3 How much faster is TESTAR when running on multiple nodes when compared to a single instance?

3. Background

To give a better understanding of the components and parts involved, some explanations will be given about Kubernetes, Azure DevOps, TESTAR and build environments.

3.1. Docker container

A docker container is a minimized image containing the software to be executed [2]. The container is built by a Docker daemon that executes the steps specified in the Dockerfile. After executing the steps in the Dockerfile, a snapshot is made and placed in a specific file format that Docker daemon can read. This file can be uploaded to a Docker repository or run locally by Docker. Docker is able to do a lot more, however, in the context of this study, only the imaging, build, and push functionalities will be used. The technical details are abstracted for the user by using Kubernetes.

3.2. SeleniumGrid

SeleniumGrid is software created to run in parallel on different browsers and operating systems[13]. There are two mainstream lines in the SeleniumGrid software. The first line is a complete setup of a grid with a central management node, and lots of workers that execute the tasks provided by the management node (called a hub). The second line is a stand-alone version of the worker. These are functional without the management node and can be customized. All these different images are created as Docker containers. In this thesis the stand-alone version of the Selenium image containing Chrome is used. Alternative stand-alone images provided by Selenium are Firefox, Edge, Opera, Internet Explorer and Safari.

3.3. Kubernetes

Kubernetes is a platform developed by Google. It is a suite with different components easing the management and complexity of large distributed systems. Kubernetes has several key points:

- Nodes
- Pods
- Services
- Ingress
- LoadBalancers
- Deployments
- Jobs

A node within Kubernetes is a processing machine that is the lowest level of a Kubernetes cluster. The nodes are internally connected with their own IP network to communicate with other nodes and present the pods with a virtual network. The nodes are responsible for the execution of a deployment. The master node determines where a specific

deployment takes place and distributes the tasks to the nodes that will deploy the required pods and other infrastructure.

A pod is comparable to a virtual machine, however it is specialized in a specific task [6]. A pod is running a Docker container. Docker containers can contain anything. An important aspect of Pods is that they do not have the ability to store data on the pod itself. All information has to be stored somewhere else. Pods use Docker images stored on a repository. There are public and private repositories, and depending on what is needed, a selection can be made.

Several pods that are doing the same task are contained in a service [7]. A service is scalable. It is possible to specify on which nodes pods have to be deployed and how many instances are needed. The service can have an internal IP or be given an external IP.

All HTTP traffic can be directed using an Ingress [5] to the outside world. The Ingress takes charge of registering requests and security.

An Ingress or Service can be exposed by a load balancer [3] to the public internet depending on the kind of traffic.

When building a new solution, a description is made of what is being deployed and how it should be deployed. This is described in a deployment [4]. The deployment contains the description of the different pods used and where to place them. It describes the services and eventually describes their IP addresses. Furthermore, the loadbalancer, ingresses and tasks are described in a deployment. A deployment is a container description of all the specific parts.

TESTAR will be deployed on Kubernetes so it is easily scalable. Before this can be done, TESTAR has to be containerized. How TESTAR can be containerized is a research topic discussed in this thesis.

3.4. Azure DevOps

Azure DevOps is a platform for building and maintaining software. The functions of Azure DevOps are to support the developer building software, plan software requirements, keep the source code, build and release the product, test the product and give information about the process. It has support for software repositories, sprint and backlog management, continuous deployment features and a test management interface. Interesting part of the build and deploy processes is that automated testing can be a part of it as well, even as deploying to a Kubernetes cluster. The backlog and the sprints are used for project management and keeping track of what code is in what story. The backlog is divided by epics, features, product backlog items (PBIs) and bugs. An epic describes a major functionality containing multiple features. A feature is made out of many PBIs containing the actual work to be done.

Azure DevOps currently uses mainly Git [9] as the source repository. Developers check in their code to Git and often relate a specific PBI, so traceability of the code is seen in the backlog items and code can be reviewed as such. There is a review process available for code reviews. However, this is outside the scope of this thesis.

One of the strengths of DevOps is that whenever a developer checks in some code, a build pipeline can automatically be triggered. For example, when a developer makes a change to Java source code and the developer checks this code into the source repository, this will be a trigger for the build pipeline. This is called Continuous Integration [14].

A build pipeline is a description of what has to be done to make the source code into

a complete product. The Java example needs a Java compiler to compile the source. If there are additional steps needed, then these too will be included in the pipeline. Finally, the result called an artifact will be uploaded to the DevOps services. Since the tasks are completely configurable, it is also possible to make a Docker image and push this resulting build to a Docker repository.

After a build has been made, it is possible to release this build to wherever you want to have it. In the case of this project, two pipelines will be made: one to build the TESTAR tool itself and push it on a Docker repository, and the second one will be made where the SUT is being built, deployed and tested by using the previously uploaded TESTAR from the repository. Possibly a third release pipeline has to be setup for the deployment of TESTAR itself.

3.5. TESTAR

TESTAR[1] is a test tooling for graphical user interface tests [15]. It does not use scripts but just performs actions on every actionable widget in the SUT. This can be typing, clicking, scrolling or other actions available to the widget. While testing the system, TESTAR creates a state model[10] with detected and visited states and every time it performs an action it investigates what actions are available and executes an action depending on the implemented action selection algorithm. TESTAR is running on a single machine and starting a browser to test a website or starts the application and starts performing all possible actions. It then reports possible faults.

Furthermore, TESTAR is able to run graphical user interface tests on applications. This means it is possible to test web interfaces as well as desktop applications.

TESTAR has a spy mode that helps the tester to identify the useful widgets and specify SUT specific TESTAR configuration. Widgets are components on the screen, for example a button, a menu item or a textbox having properties.

TESTAR is written in Java and built against the JDK (14 is working at the moment of writing). It is setup using several libraries providing functionalities for TESTAR. TESTAR can run with or without maintaining a statemodel. When the state model is used, TESTAR uses a graph database from SAP called OrientDB. A graph database is a database that has nodes, vertices and edges. In TESTAR, it will be used to store the available actions on the edges, and the states as vertices.

4. Related Work

There are several theses already written on the TESTAR tool and how it is applied in industrial environments. Furthermore, there are publications about the integration of TESTAR in a continuous integration environment.

4.1. Inclusion in Continuous integration framework

In this study, TESTAR will be part of a continuous integration pipeline. Research has been done about this topic by Aho et al [1]. The way this has been tested was by integrating the building of the TESTAR tool just before it needs to be executed. When built, it was executed on the SUT. Reports are collected and sent to the developers. The way this project will continue is build the project (but not continually integrate the newest version), execute the

[1] testar.org

tests on a Kubernetes cluster that collects the results of the different nodes and reports the results back to the DevOps platform using a generalized error handling and reporting using of the NUNIT 3 standard. The related research involved the inclusion of TESTAR with the Jenkins deployment tool [12]. This however was not tested in a parallel environment like this study

4.2. Distributed GUI testing

There is a patent on distributed GUI testing[8] describing a method to distribute tests over more than one server. The solution proposed in this thesis is different to the patent because the method proposed by the thesis is to gather all available actions and add those to a queue. The nodes should then get the first available action from the queue. The solution the author proposes is without the use of a queue. Instead, a graph database will be used. An additional important difference is that there is no distribution of test methods. The patent describes a method where a controller distributes the test to the agents. In this research, the agents will retrieve the available states and select an unvisited one to get new states. This is the push versus pull principle.

5. Research and problem analysis

When investigating the possibilities to integrate TESTAR in the Continuous integration/-Continuous delivery (CI/CD) pipeline of the case study company a few specific things need to be done. These have partly to do with the SUT and partly to do with the delivery process of the software.

The goal is to make sure that: ”TESTAR needs to be able to perform all the tests on the SUT in parallel sharing the state model on a Kubernetes cluster and be able to report the results back to the release pipeline itself. To perform tests on the SUT it needs to be able to perform a registration process as well with complex forms and a lot of steps.”.

5.1. Research questions

The goal of this research is to find answers to the following questions.

- "RQ1 How can TESTAR perform all the tests on the SUT in parallel on a Kubernetes cluster"
- "RQ2 How can TESTAR be integrated in a CI/CD pipeline with the specific SUT?"
- "RQ3 How much faster is TESTAR when running on multiple pods when compared to a single instance?

5.2. Problem analysis and preliminary research of RQ1

Testing with TESTAR takes possibly a long time which will make it hard to be used in daily builds that are run early in the morning and should be completed before starting to work. When running TESTAR and willing to find errors it should run for a long time.

The solution proposed in this study is to parallelize the testing with TESTAR where every pod [6] visits unvisited actions and doing so optimize the testing time. TESTAR has the ability to use a state model [10] to store all states coming from the previous state, the available actions and the state transitions. More information about this subject is given in the section 5.2.2 about the state model.

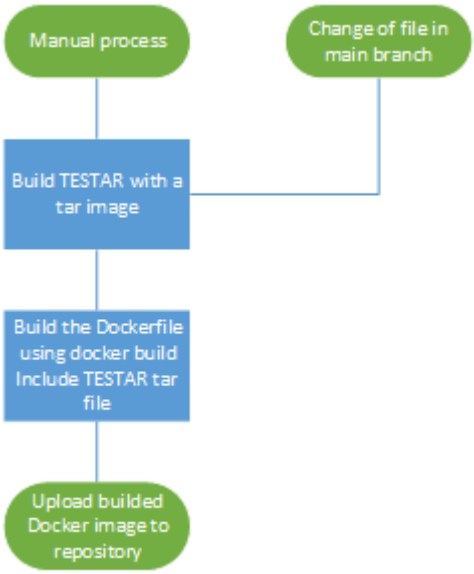

Figure 1: Build process

From the statement taken from the introduction of this section, several actions need to be taken to achieve the desired results related to RQ1:

- Containerize TESTAR into a **docker image**
- Create the ability to **share the state model** between different pods
- Implement an **action selection algorithm** so the different pods select the right actions

Each of these are described in the subsections below.

5.2.1. DOCKER IMAGE

There are several requirements for the Docker container that has to run TESTAR. The Docker image needs to contain a Chrome version. Furthermore, the Selenium Chrome Driver should be there as well. This Web Driver is required by TESTAR to provide instructions to Chrome. The configuration of the image must be provided. TESTAR requires these files on a certain location. TESTAR itself needs to be compiled and being put in the image and the image has to be uploaded to a central repository. The process to get TESTAR in an image is preferably automated and run from a build task on a checkin on TESTAR as shown in figure 1. Alternatively a build of the Docker image can be done manually.

The initial investigation during the VAF of the docker base image was started by using an alpha version of the SeleniumHQ Grid software. The Selenium HQ Grid team created an image that was already containing all the software required. The image contained Chrome, an X-server and a VNC server (the X-server and VNC server are required so an external machine can connect with the instance and see the screen of the pod where TESTAR is running) so it was an easy method to extend this and install the TESTAR software on the image.

The problem with the image being a part of the grid from SeleniumHQ was that the amount of change of the software components on that particular Docker image was very

high. This however was specifically for this container which main purpose was being a part of the Selenium grid. The alternative image that contains the standalone version of Selenium including a Chrome browser is more stable, basically meaning that the amount of software components being added or removed is not happening often.

5.2.2. Share the State model

TESTAR, in the beginning of this research, did not retrieve the states from the state model before making another action. The observed behaviour is that TESTAR loads its initial state model into memory by retrieving the states from the database in the beginning of a test run. During execution, TESTAR adds new states to the database. However, states are not loaded from the database during the execution, so multiple TESTAR instances running at the same time would not benefit during runtime from each others data that is added to the graph database. During the VAF the first step was to investigate whether it was possible to load the states during execution before action selection. To be able to share the states with all the different pods, a central database is needed. Every pod determines its current state and makes an abstract state id from its current state. TESTAR then checks all the different actions that are available and stores them as well. TESTAR supports the use of a Graph database. This graph database stores edges and vertices from the state model[10]. In the preliminary research done during the VAF, it was observed that several tables are relevant. The relevant tables are the AbstractState and the BlackHole vertices as shown in figure 2. In the AbstractState table are all discover states stored. Furthermore the edges are important. Relevant for this study are the UnvisitedAbstractActions and the AbstractActions. The vertices from AbstractState to AbstractState are connected with AbstractAction edges. The AbstractState is connected to the blackhole using an UnvisitedAbstractAction. In this research the main emphasis will be on how to visit as fast as possible all the unvisitedabstractions. The tests are considered done when all discover actions are performed at least once and the state model does not contain unvisitedabstractactions anymore.

During the VAF a research was carried out how to create an method that retrieves all the unvisited actions from the database. At that moment it was seen that the ActionSelector must be modified. During this thesis the author came to the conclusion that only access to the database is needed from the protocols. The protocols are descriptions of how a specific application should be tested. When making the database session available in the protocols it would be possible for a protocol to perform its own queries against the database and so make decisions based on data of the database.

5.2.3. Action Selection Algorithm

An algorithm needs to be devised to base the selection of the action on a shared state model. There are several requirements for this algorithm:

- From every state the algorithm should come closer to an unvisited abstract action
- Two pods shouldn't be heading for the same action
- The pod should always be able to find a shortest path to the next unvisited abstract action
- When a pod can not reach an unvisited action using a shortest path, a shortest path probably doesn't exist and a new sequence will be started

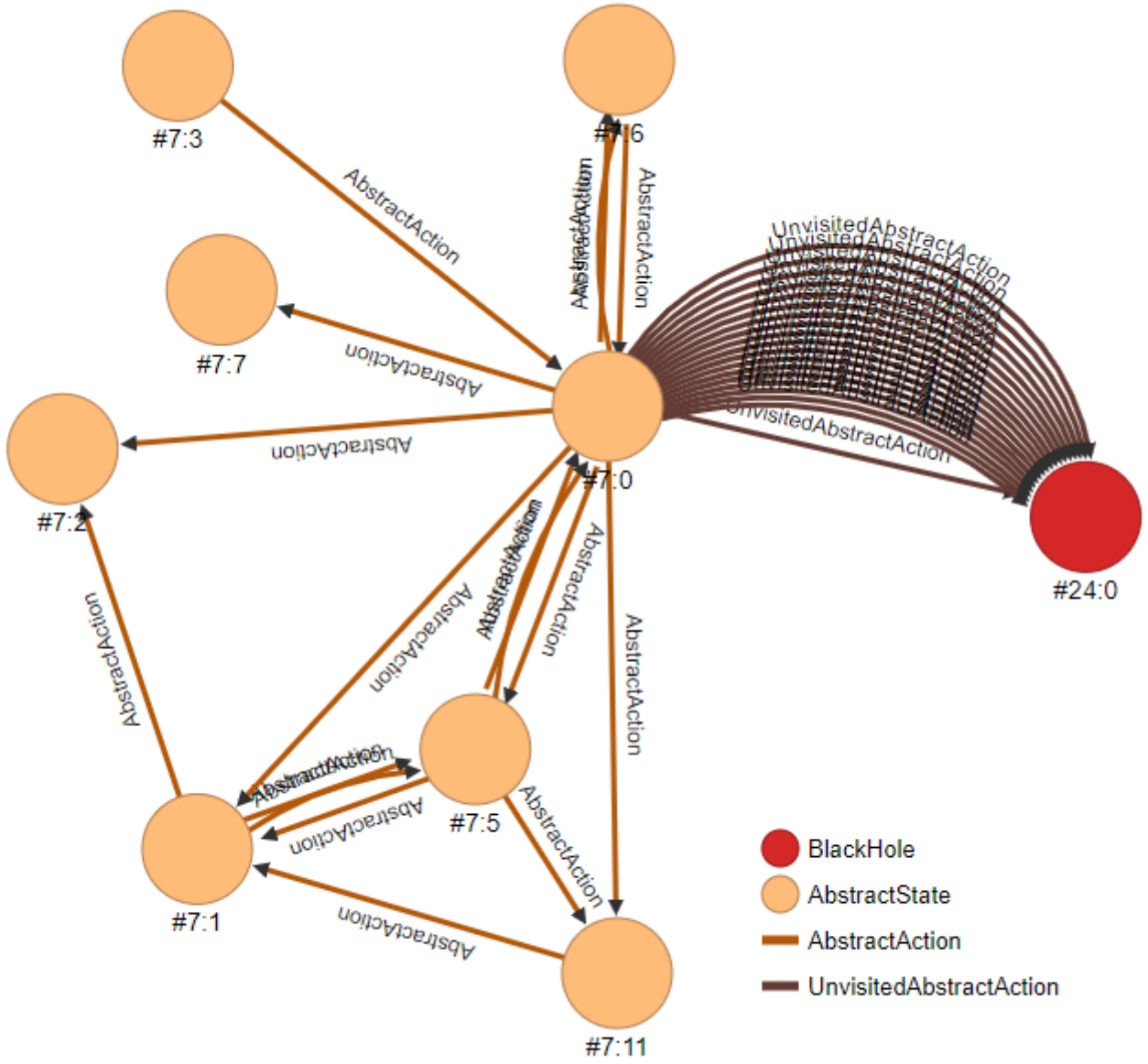

Figure 2: Example of state model

5.3. Problem analysis and preliminary research of RQ2

The second research question RQ2 is about how TESTAR can be integrated in a continuous integration pipeline of the SUT. For this the following things need to be performed:

- Describe an infrastructure file to be **deployed to Kubernetes**
- Implement a **logging facility** that is able to report back to the pipeline
- Implement a method to **fill web forms** on the different pages (this is a requirement for the SUT)

Each of these are described in the subsections below.

5.3.1. Deploy on Kubernetes

To answer the second question the application needs to run on Kubernetes using Docker images. It is very interesting to see that this is possible, however changes to the infrastructure have to be made. Running with multiple pods brings new opportunities but also pitfalls. For example, when integrating TESTAR into a build, it should be run as a task that returns after a certain time with the results to report to the DevOps release pipeline. Every time a deployment takes place, it should setup an OrientDB database with possibly (this is a choice) a persistent storage. However, it is a choice of the company whether the tests should start from scratch or continue the previous tests. So, to be able to do a complete automated test run a deployment should be made with the following components:

- the OrientDB
- a specifiable amount of pods running TESTAR
- a place to store the log files, configuration files and protocol file

In the previous section it was described how the image should be made and what the requirements of the image are. A choice can be made whether the OrientDB will be a Docker image and run on the Kubernetes cluster. In this case study a Docker image is preferable due to the fact that the database will only be there during the testing of the SUT. After testing the SUT the complete environment can be removed. In this study the specific results will not be available anymore for future testing. This is as well the reason to use non persistent storage for the logging and deploy them as well as a Docker image. Testing the SUT is considered finished when all unvisited actions are visited so no new states are found.

Next to this the SUT itself needs to be deployed as well. Since the SUT is probably changed and will be rolled out to the test-environment this needs to be done first. As soon as the roll out completes, the software needs to be tested. This is the place TESTAR kicks in as shown in figure 8.

5.3.2. Logging

At this moment TESTAR is able to log using an HTML format. HTML is not meant as a format that could be used for interprocess communication. Interprocess communication at this point means the communication of the results of the test towards a test engine that can read the results. In this thesis an implementation is done of the NUNIT 3.0 format that could be read by the test engine of Azure DevOps. Test results from TESTAR will be available in the Azure DevOps after the files are uploaded.

5.3.3. Form filling

Currently TESTAR is having difficulties to complete a registration on a portal with a lot of fields. When considering n fields and having a next button to continue, then most simply said by just considering whether the field will be filled or it wont, there is a possibility of $\frac{1}{2^{n+1}}$ that TESTAR will complete the form at this moment. Due to the immense amount of possibilities to fill a field or leave it, it would be practically impossible to complete a large form and fill it correctly. By correct in this sense it is not only that a value has been filled in but also that field contains a context related value. A random value in such cases will often not be allowed (for example postal code, IBAN number or tax number validation). TESTAR has to be adapted in such a way that it can recognize a form in a web page, and from that point, complete it as a single action. In this way, complex form filling can be handled. Validation can occur by testing the page of the SUT and see if it is able to fill the page. After it fills the page, it can click on a continue button to go to the next screen if the action is performed. There are a lot of ways to do this form filling. The method chosen in this thesis is that forms will be automatically detected. Then TESTAR will check whether there is a stored file for this form and the content of the file will be used to fill the form. Alternatively, if it does not exist, a template file will be generated so that after a run the user could supply the values for future runs.

5.4. Problem analysis and preliminary research of RQ3

RQ3 is about measuring the performance and finding the limits of the amount of TESTAR pods deployed. The author expects that adding more pods will at a certain moment stop its effectiveness. Effectiveness in this sense can be described as the moment when adding more pods will not significantly impact the time the system needs to be tested. As earlier mentioned the system is considered tested when there are no unvisited actions anymore. For example, when 1 pod takes 1 hour, 2 pods take half an hour. 4 pods take 15 minutes, but 8 pods take 11 minutes (and 16 pods take 10 minutes). When looking at the difference in testing time needed it can be seen that adding 4 additional pods will not add a lot of difference to the time needed for measuring. Validation of this part will take place on the Parabank test website hosted on testar.org.

The method being used is as follows:

- Test the containerized TESTAR from section 5.2 and 5.3 on the website Parabank
- Verify that the algorithm works as well on the SUT in the company

The algorithm will be described in section 5.2.3. The experiment that is performed will be several iterations of the same software using different amount of pods executing the tests. The algorithm for action selection and form filling will be available in TESTAR by using a protocol file and every time the same protocol file will be used. The experiment will be started with one pod, and the time it takes to discover all the different states will be recorded as well as the amount of found states. This test will be repeated for several times on each pod to get an average time it takes to discover all the states. The average time and the variance of the time will be recorded as well. This will be the start position. From start position the same test will be repeated on the same hardware but then using two pods. The same results will be recorded. Additionally from the second pod till the last pod a calculation is made about how much faster the increase of pods will be regarding the

previous amount of pods. The expectation is that adding more pods will in the beginning show drastic improvement, but when adding more pods, at some point it will no longer significantly reduce the time the tests take to complete. So from one to two pods the results will be improved, but for example from ten to eleven pods will only improve by for example 3 % in total speed. This means that adding more pods from a certain moment will not make much sense anymore.

6. Research Implementation

6.1. RQ1: Parallel testing with TESTAR on Kubernetes cluster

RQ1 How can TESTAR perform all the tests on the SUT in parallel on a Kubernetes cluster

Remember from section 5.2 that to answer RQ1, the following actions needed to be resolved:

- Containerize TESTAR into a **docker image**
- Create the ability to **share the state model** between different pods
- Implement an **action selection algorithm** so the different pods select the right actions

We will discuss the outcomes of each of them in the subsections below.

6.1.1. Containerize TESTAR into a docker image

In the preliminary research (VAF), it is described that the container from Selenium is not stable due to the many software component changes in the container. This was mainly caused by the fact that the maintainer removed the complete VNC server part. From this perspective the author made the conclusion that the image provided by Selenium was unstable to use and extend. During this thesis, a new perspective has been gained and it has been implemented. The selenium image that is not part of the grid but running as a standalone is used as a base for the Docker image containing the TESTAR build. Selenium maintains different images for their software to be containerized. In the preliminary research, a Docker container was used which is used in a node and not meant to be used as a standalone container.

The first plan was to create a fully headless Docker image with TESTAR also being able to run headless. The problem encountered was that at the current state Google Chrome will not support extensions in headless mode. According to the Google team they will not implement this either due to the large amount of refactoring required.

However, the Selenium image has a framebuffer version that could be used. A framebuffer is used as a virtual screen in memory where the browser will be displayed. It takes more memory than native headless where the content is not even displayed in memory. A framebuffer is often slower since all rendering takes place in memory even though this is not displayed anyway.

Docker container setup

The container is built as described in source listing 1. TESTAR is already compiled in a TAR file before starting the build of the Docker image. The location is relative to the Dockerfile. Currently this is situated in the root directory of the source repository.

The base of the container is Selenium Chrome as seen on line 1. Lines 4-6 describe the required components to run TESTAR. On line 8 TESTAR is extracted inside the image. Lines 10 and 11 contain environment settings. Line 13 takes care of copying the script to TESTAR to the image Line 17 takes care of starting the script to run TESTAR

This image can be build and pushed towards an Docker repository, so it can be executed by other nodes that run on Docker based containers.

Listing 1: Dockerfile

```
1  FROM selenium/standalone-chrome
2
3  USER root
4  RUN apt-get -o Acquire::Check-Valid-Until=false \
5      -o Acquire::Check-Date=false update \
6   && apt-get install -y openjdk-14-jdk libxkbcommon-x11-0
7
8  ADD testar/target/distributions/testar.tar .
9
10 ENV JAVA_HOME "/usr/lib/jvm/java-14-openjdk-amd64"
11 ENV DISPLAY=":99.0"
12
13 COPY runImage /runImage
14 COPY README.Docker /README.Docker
15 RUN chmod 777 /runImage
16
17 CMD [ "sh", "/runImage"]
```

In listing 2 a description is given about the script that is executed when starting the image. It starts by displaying README.Docker giving general instructions about how to use the image. Then it starts the framebuffer and finally starts TESTAR.

Listing 2: runImage

```
#!/bin/sh
cat /README.Docker
echo "Start Xvfb"
/opt/bin/start-xvfb.sh &
sleep 2
cd /testar/bin
./testar
```

The container has been set up in such a way that it can use configuration from a central location (by mounting the settings directory to a specific shared location) so that all TESTAR nodes can use the same configuration. Logging must have a central location as well and can be mounted the same way. When starting the TESTAR docker image, it will gather the required configuration from the location /testar/bin/settings in the image. When starting the image files and directories can be overriden, so a different configuration can be

used by binding another location to this directory of the image. In a later stadium we will see this is required for running it on a Kubernetes cluster where no information is provided and customizations to the configuration are required. If the Docker image is setup correctly it will show the welcome text about how to run and will start running the tests immediately.

This Docker image can be run by Kubernetes and Docker. During the application of this Docker image in a real environment Kubernetes would be used to deploy the software on. Advantages of having TESTAR as a Docker images is that it is very easily deployable.

When this part is running, TESTAR will be able to run in a Docker container using a standard shared configuration between nodes and log to a central location. When the configuration that is used to start the Docker container does not contain a state model configuration then TESTAR is executed without a state model. The state model is not a requirement to be able to run Docker in a container. However for this study a state model is needed. During the tests a standalone docker image of the OrientDB was used. The Docker image executed was

```
docker run -d –name orientdb -p 2424:2424 -p 2480:2480
-e ORIENTDB_ROOT_PASSWORD=root orientdb:3.0.38
```

Integration of Kubernetes in the pipeline

When having an infrastructure file it is possible to integrate this into a deployment pipeline. When deploying a website additionally the infrastructure can be deployed to the Kubernetes cluster and the TESTAR software will start testing the website immediately. The next task of the deployment pipeline will be to wait till the task completes or the timeout expires. When the timeout expires the task should stop so the release pipeline is available again and the results can be gathered from the configured output directory and pushed to the DevOps environment.

The release of new software has also been shown, also using Azure DevOps by the use of the release pipelines. A sample working pipeline can be seen in figure 4 The SUT is deployed in the first place to the target system. In this image the start of the deployment went with the creating of the database script. Then the database was deleted and a new one was created with the generated script. Next step is copy the protocol settings file and test file to the cluster. After this copying the Kubernetes cluster gets the instruction to run TESTAR using the specified nodes in the configuration. The amount of nodes can be selected in the parameters as shown in figure 3. Afterwards the Kubernetes implementation is deployed on the kubernetes cluster that can start testing the SUT using TESTAR using the specified amount of nodes. The next step is the wait for the completion of the cluster.

6.1.2. Sharing state model between instances

A new algorithm is implemented during this research to select actions by using a shared state model between the different instances working together on that state model. Goal of the algorithm is that the number of unvisited abstract actions goes down to zero as quickly as possible. Before discussing the algorithm some explanation of the state model should be done.

The state model (an example shown in Figure 2) consists of edges and vertices of different classes. The edges in this algorithm are labelled with two types: AbstractAction and UnvisitedAbstractAction. Both of these edges are interchangeably called action. The vertices are labeled in three types and called BeingExecuted (newly introduced for this study),

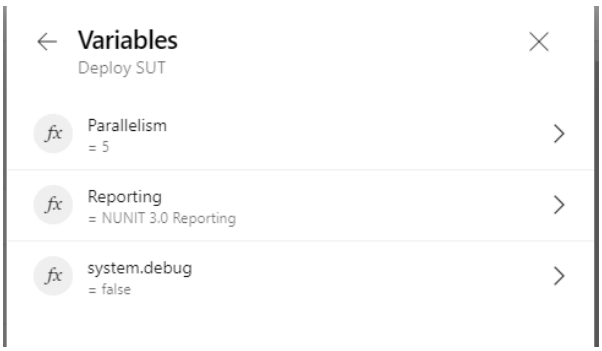

Figure 3: Configuring variables in Azure DevOps

Algorithm 1 Random unvisited action selector from database

```
1: function SELECTACTION(currentState, Set<Actions> availableActions)
2:     if selectedAction == null then
3:         selectedAction ← selectNewAction (currectState, availableActions)
4:     end if
5:     if selectedAction ∈ availableActions then
6:         action ← availableActions[selectedAction]        ▷ Action currently available
7:         selectedAction ← null
8:         return action
9:     else
10:        action ← TraversePath(selectedAction)   ▷ TraverseAction algorithm described
   in other location
11:        return action
12:    end if
13: end function
```

BlackHole and AbstractState. The vertex BeingExecuted is used by a node to reserve a specific unvisited action that it wants to execute. The BlackHole is a vertex where all discovered but not yet visited actions are pointing to. An AbstractState S can have 0 or more unvisited abstract actions A. All unvisited abstract actions (A) go to a single instance of the BlackHole. In figure 5 this is shown by the orange vertex (AbstractState) leading to the red vertex (BlackHole). When an unvisited action is performed by the browser this will become an abstract-action AA with an updated end of the edge going to an AbstractState instead of the BlackHole.

6.1.3. THE ACTION SELECTION ALGORITHM

From section 5.2.3, remember that the algorithm has the following requirements:

- From every state the algorithm should come closer to an state where an unvisited abstract action is available
- Two pods shouldn't be heading for the same action
- The pod should always be able to find a shortest path to the next unvisited action
- When a pod can not reach an unvisited action using a shortest path, a shortest path probably doesn't exist and a new sequence will be started

Algorithm 2 Select new action algorithm

```
1: function SELECTNEWACTION(currentState, Set<Actions> availableActions)
2:     availableActionsFromDb ← select shortest paths from currentState to BlackHole
3:     if availableActionsFromDb.size() >= 1 then
                                   ▷ A shortest path to an unvisited action exists
4:         action ← select random action from availableActionsFromDb
5:         return action
6:     end if
7:     countOfAbstractStates ← count abstract states in database
8:     if countOfAbstractStates = 0 then
                          ▷ Select random action since database is not yet initialized
9:         action ← select random action from availableActions
10:        return action
11:    end if          ▷ If we arrive here no actions do exist in the database at all
12:    action ← Select random action from availableActions
13:    stop ← true                                          ▷ Stop all sequences
14:    return action
15: end function
```

Algorithm 3 Traverse path algorithm

```
1: function TRAVERSEPATH(currentState, Set<Actions> availableActions)        ▷
   Method to select a new action
2:     stateFromAction ← select stateId from the abstractstate that is the origin of the selectedAction
3:     path ← select shortestPath from currentState to stateFromAction      ▷ Path is
   array of vertices that need to be visited
4:     if path.length() < 2 then     ▷ If there are no two vertices connected using abstract
   states path traversal shouldn't have taken place. Stop the sequence all together
5:         moreActions ← false
6:         return random action from availableActions
7:     end if
8:     abstractAction ← Select from abstractAction where in = path[0].rid and out = path[1].rid
9:     return aabstractAction
10: end function
```

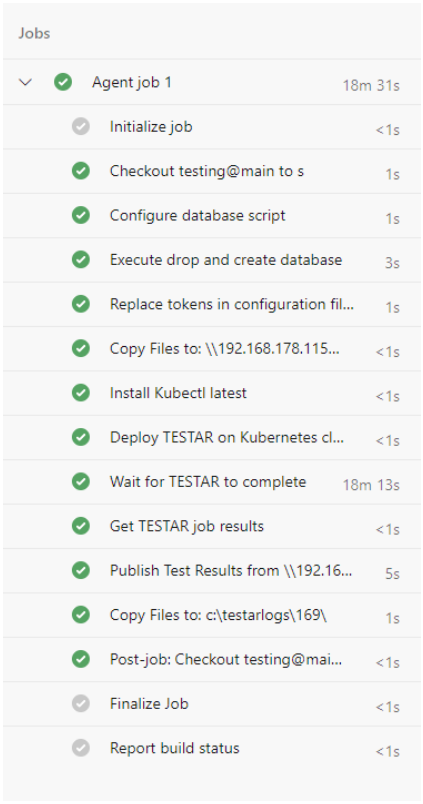

Figure 4: Run of an Azure build including TESTAR

The actions that need to be performed are the unvisited actions. An action is visited when the action is performed in the browser and this information is stored to the database or internal model. (For this algorithm, only the database is relevant). In figure 5 is shown how abstract states (orange), beingexecuted (green), the blackhole (red) are related to each other using the abstract action and the unvisited abstract action.

The algorithm should work as follows and is described by a pseudo-algorithm in Algorithm 1:

1. If currently no action is selected, select a new action by choosing a shortest path from the current state to the blackhole. If no shortest path is available, check whether the amount of abstract states in the database is zero (This means that the database is empty and possibly not fully initialized). Select a random action. This SelectNewAction algorithm is described in Algorithm 2. If there are no actions since all actions are visited, we can stop the test run.

2. If an action is selected then two options are possible: The action is actionable from the current state so it can be performed. Otherwise the complete path should be traversed till the action is there. This is described in Algorithm 3.

6.2. RQ2: Integrate TESTAR in CI/CD pipeline

RQ2 How can TESTAR be integrated in a CI/CD pipeline with the specific SUT?

As discussed in chapter 5, to be able to test the SUT and integrate it in the pipeline three features have to be implemented:

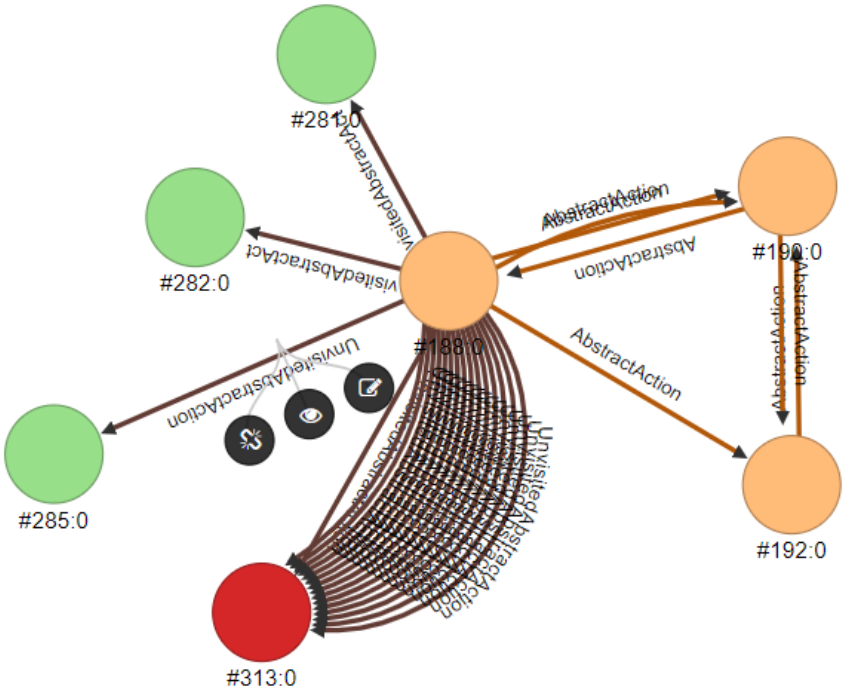

Figure 5: Traversing the paths to an action

1. Logging facilities to be able to integrate in a DevOps CI pipeline,
2. Creating a Docker container in such a way that it can be running on Kubernetes.
3. The ability to handle form filling to be able to test the SUT of the company

This order is chosen so all the specific parts of the implementation will be done in such an order that all dependencies are done before starting the next part. For example, adding logging is not dependent whether the software is running in a container. However, implementing the part where the container will test the software without being able to log in the correct format will result in an container delivering an unreadable log for further processing.

Since the SUT has a lot of fillable webforms where functional context of the system has to be known, the feature for form filling needs to be implemented as well.

6.2.1. IMPLEMENTING LOGGING

Logging is implemented using the NUNIT 3.0 specification [11]. This specification is a form accepted by the Azure DevOps pipeline. NUNIT 3.0 is furthermore standardized and usable in many other deployment environments as well.

TESTAR contains a Java class HtmlSequenceReporter, as described in Figure 6, that is able to report a sequence with a list of actions in HTML format. Looking at the HtmlSequenceReporter, the public functions that are contained in this class can be abstracted to a Java interface so other implementations with the same functionality can be built. TESTAR uses the HtmlSequenceReporter class for many different protocols. Refactoring the complete interface is not feasible since it is used on many places. The alternative to this is to create a Java superclass of the HtmlSequenceReporter (now implemented as the interface

Reporter) that is an Java interface containing all the public functions the HtmlSequenceReporter contains. From this point everywhere in TESTAR where the HtmlSequenceReporter is used, it can be replaced by the more generic Reporter interface. The instantiation of the HtmlSequenceReporter will be done using a generic method implemented in the AbstractProtocol. The default protocol is extended with a Java function getReporter() that uses the loaded TESTAR settings to determine the Reporter that should be instantiated. The configuration is extended as well to include a ReportingClass tag. This tag allows the user to save the format to be used with the protocol.

The new NUNIT 3.0 reporter is of the earlier described Java-type Reporter. The new Java class is called the XmlSequenceReporter. In this XmlSequenceReporter class the NUNIT specification is implemented. The NUNIT specification describes an XML format containing a root element called test-run that is the main node. The main node contains several nodes called test-suites. The different test-suites contain multiple test-cases. On every node of every type the sum is shown of its children. Information registered in the different nodes is the amount of test-suites, test-runs, failed test cases, successful test cases and skipped test cases. For every TESTAR sequence a new test-suite is added to the test-run. For every selected action by TESTAR an test-run is added to the test-suite. The action is always successful unless the verdict provides an error. In that case the test-run is reported as a failure. After the implementation of logging is completed it possible to upload the results are being uploaded to DevOps so they are available for the end user. This can be configured in the pipeline so the test becomes an part of the build (or release depending on what stage TESTAR is applied. The logs are available now in Azure DevOps as seen in figure 7.

Another important part of integrating TESTAR in the pipeline is the ability to report to Azure DevOps using a format that could be interpreted. The author implemented NUNIT 3.0 to store the test results in such a way that the results could be read by Azure DevOps. From the moment the results are in Azure DevOps it is possible to gather information about test results from the Azure DevOps system.

6.2.2. FORM FILLING

TESTAR is able to fill some fields, however the ability to fill a complete form when performing actions randomly is hard to achieve. Chances that n fields are filled with only the assumption that the field will be filled or it will not and having a submit button that can or will not be clicked means that there is a chance of $\frac{1}{2^{(n+1)}}$ that the form will be completely filled in and is submitted. TESTAR has protocol drivers that have several hooks where actions can be performed. The hooks are used to specify actions, perform actions, select actions, determine whether there are more actions for the sequence and determine whether there are more sequences.

To be able to fill a form the author used the deriveActions function hook that allows to create actions and add those to a set of actions. The algorithm the author implemented is described as follows:

- Detect all form tags of the current state of a website
- When a file exists containing the location of the page in combination with form id, load this file
- When a file does not exist, record all fields so they can be stored into a template file, otherwise if the file exist read all the values from the file and use the value from the

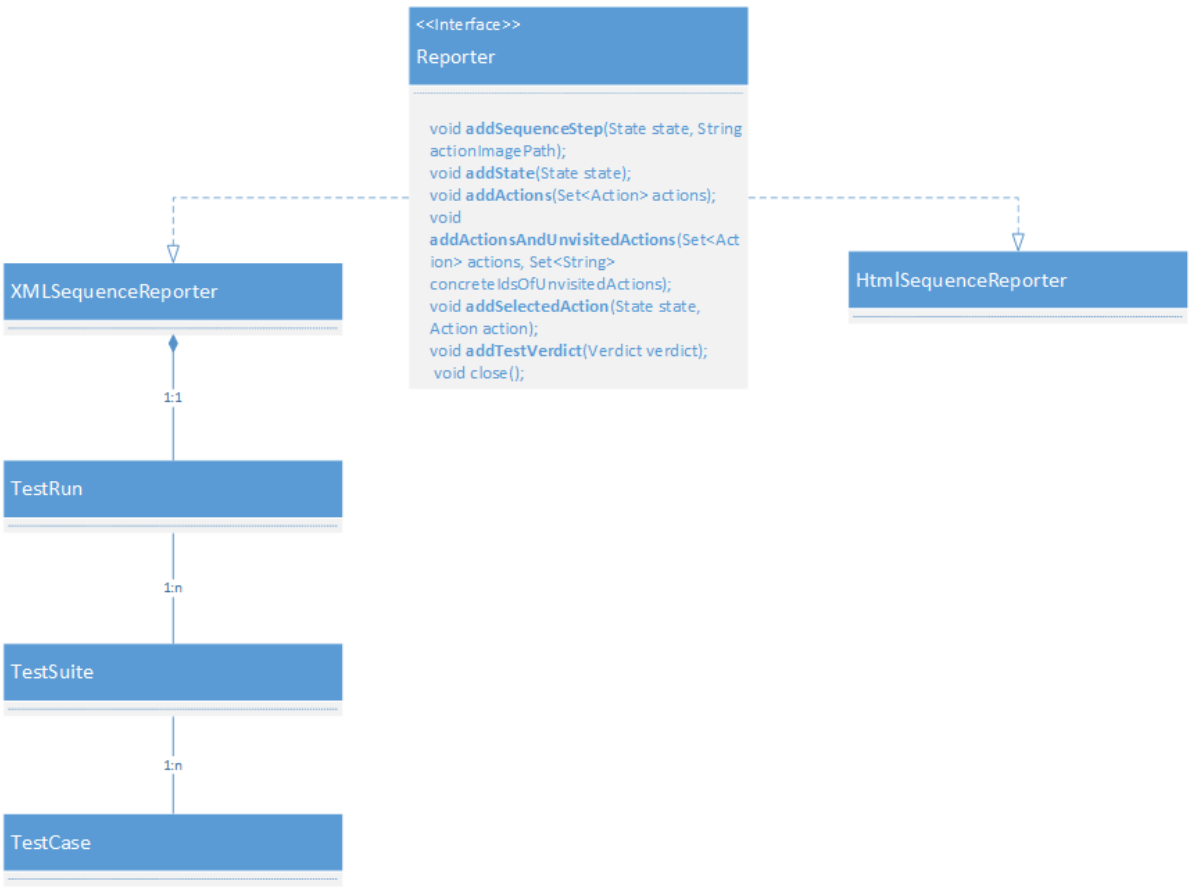

Figure 6: Redesign of reporting functionality in TESTAR

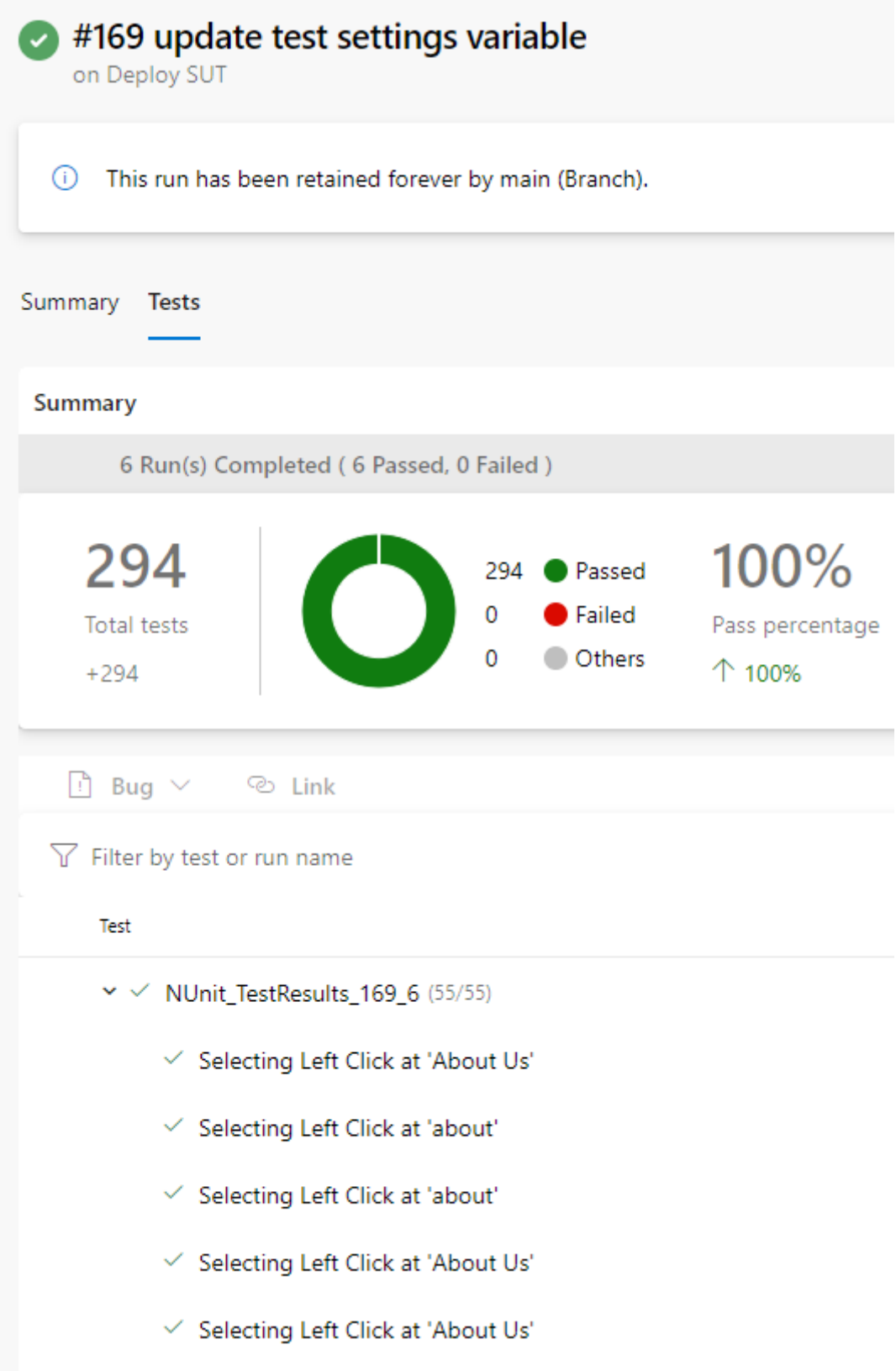

Figure 7: Test results shown in Azure DevOps

loaded file

- Submit the form

The result of the algorithm is a CompoundAction. A CompoundAction is an action that is built out of multiple separate actions and combined into one. When a form is seen the algorithm will traverse the complete form. Within an HTML form, there can be a lot of nested elements that are not typeable by itself, for example a table with input fields. In such a case the form element is seen, whereas the following element is a table. The table having rows and the rows possibly have columns. Then on every column it would be possible to have a field that can be filled in. This is an example of a nested structure where the algorithm should make a combined action from all of these nested inputs and make it a single executable action for the test software. The algorithm for this reason is recursive. A simplified version of the algorithm is shown in Algorithm 4. The complete algorithm is in Appendix B (in Chapter 8.2) where the complete protocol file is added. The function is started. The fields parameter is used for when there is already a map with keys combined with values about how a form should be filled in. If the form has a name there will be looked for a file on disk and those values will be read. This is done in line 5 and 6. Line 8 builds the complete form starting with the main widget and recursively go through all elements. In line 10 the form is added to the actions available for selection.

6.2.3. Implementing the solution on Kubernetes

After the containerization is done, the container including an OrientDB image will be installed on a Kubernetes cluster as seen on Figure 8. A specific configuration is made so the infrastructure will be rolled out at once. The TESTAR image is pushed to a central Docker repository so Kubernetes can download it when needed as described earlier in Figure 1 about the build process. The description of the Kubernetes deployment describes the following parts:

- Deploy OrientDB image
- Deploy OrientDB service so the service can be used
- Deploy a persistent volume for the settings
- Deploy a persistent volume for the output
- Provide a persistent volume claim to the persistent volume settings for the pods
- Provide a persistent volume claim to the persistent volume output for the pods
- Deploy a task called TESTAR having a degree of parallelism specified in combination with the persistent volume claims and the image

An example of such an infrastructure description is in Appendix 8.1.

When the deployment is applied as an infrastructure file to the Kubernetes cluster all above items will be immediately provisioned if they do not exist. The Kubernetes cluster will take care of the deployment of the different pods (containing the TESTAR image) and TESTAR will start testing. In Appendix 8.1 there is shown a section related to TESTAR in the **kind** of a **Job**. The parallelism is specified in here and defines the amount of workers that

Algorithm 4 Form filling algorithm

```
1: function FILLFORM(actions, fields, widget, storeFile) ▷ Method to create an action that fills an form
2:     if field = null then
3:         fields ← new HashMap<String,String>
4:     end if
5:     if widget is form and has a name and a file exists by that name then
6:         fields ← Read key value pairs from file
7:     end if
8:     int sum ← buildForm(widget, fields, storeFile, form)
9:     if sum > 0 then
10:        actions.add(form)          ▷ actions are the available AbstractActions that are selectable in the selectAction agorithm
11:    end if
12:    if storeFile then
13:        StoreFile(form)
14:    end if
15: end function
16: function BUILDFORM(widget, fields, form)
17:    int sum ← 0
18:    if widget is typeable then
19:        if not fields.Contains(widgetName) then
20:            fields[widgetName] ← "write-random-text"
21:        end if
22:        forms.add(widget, fields.get(widgetName))
23:    end if
24:    for all child ← widget.children() do
25:        if child is typeable then
26:            forms.add(child, fields.get(child.widgetName)
27:        else
28:            sum ← sum + 2 + buildForm(actions, fields, child, form)
29:        end if
30:    end for
31:    return sum
32: end function
```

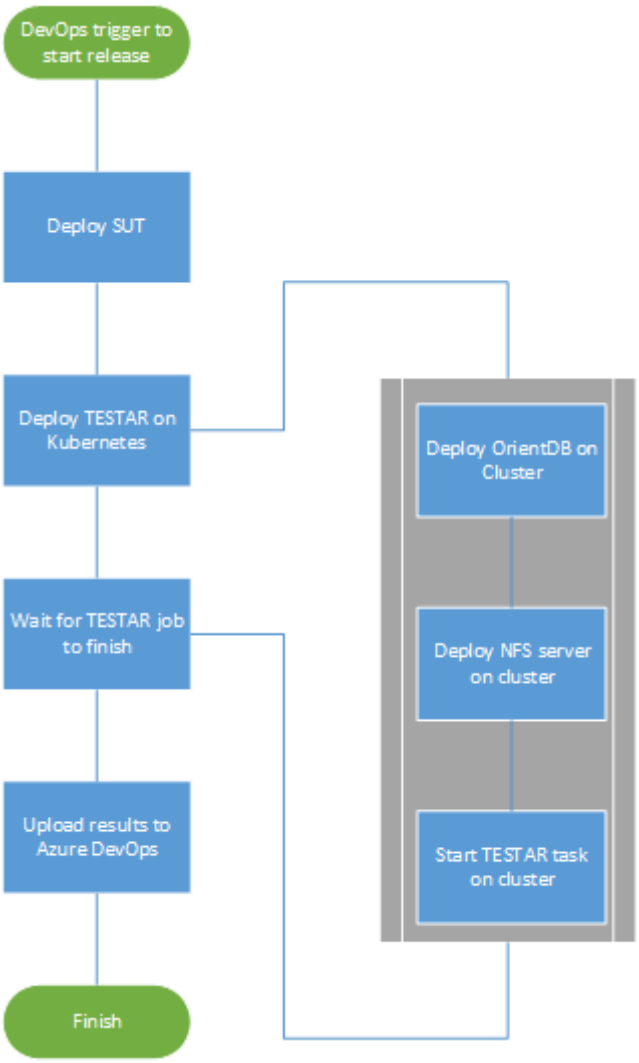

Figure 8: Deployment process

are going to execute the job. Furthermore, the Docker image containing the TESTAR image that is used is specified here. This image is retrieved from the Docker hub. The Kubernetes cluster automatically places these pods on nodes of the cluster itself.

6.3. RQ3 PERFORMANCE OF PARALLELIZATION OF TESTAR

RQ3 How much faster is TESTAR when running on multiple nodes when compared to a single instance?

TESTAR was already able to run in parallel. Adding the possibility to share states between the different nodes is added. A shortest path algorithm is implemented in RQ1 where a central database keeps track of all the different states and the visited and unvisited actions. The research is performed on a virtual machine running Ubuntu 20.04.2 with 24 gb of memory and 16 cores. The database was running in a Docker container as well. The SUT tested is Parabank and amount of states is greatly reduced so the state model could be inferred and analysed in a decent time. The amount of states in the model was 10 and the amount of edges leading from the states to other states was 131 meaning that there are 131 possible actions in the limited model resulting in 10 different states.

For the integration of the Azure pipeline Kubernetes Minikube was setup on the virtual machine. However testing was done directly by using Docker. When normally running with a pipeline the software is build, and then released. During the release the software is tested and the results are being uploaded to Azure DevOps. In this case we wanted to run Docker image many times after each other, just measuring time. Azure DevOps is possible to do this but needs a trigger. Then it would build the SUT and deploy the SUT whereafter the test will take place. To get the results it is unfeasible to constantly create trigger for a new build then setup the complete environment and run the test. So for gathering the results a script was created and executed. This script is able to run from a specified amount of concurrent node to 1 node. Then testing every configuration of nodes for 10 times. Record the time of every run in milliseconds for that specific configuration and continue with the next run. Using this method there is no trigger needed and the tests can be performed faster. Every test run the database of TESTAR needs to be deleted and created again. The algorithm is setup in such a way that as soon as it detects abstractstates and the amount of unvisited abstract actions is zero it will stop. Due to this behaviour a database can only be used one time. The bash script being used and can be performed on a machine having Docker installed:

```
#!/bin/bash
x=$1
echo . > results.txt
echo "Nodes;time" > results.csv
for ((c = $x; c>0; c-- ))
do
        for ((tst = 0; tst<10; tst++ ))
        do
                echo "Test $tst met $c nodes"
                pids=()
                echo "Drop database"
                docker run --mount type=bind,source="/home/arend",
```

```
                target=/tmp orientdb:3.0.38 bin/console.sh /tmp/command
                sleep 5

                tijd=`date +%s%3N`
                echo "Use $c nodes om TESTAR te draaien start tijd = $t
                for ((nodes = 0; nodes<$c; nodes++ ))
                do
                        docker run  --shm-size=512m --mount \
                        type=bind,source="/testard/settings" \
                        ,target=/testar/bin/settings --mount \
                        type=bind,source="/testard/output", \
                        target=/testar/bin/output \
                        aslomp/testarstatemodel:latest > uitvoer_$tst_$
                        p=$!
                        echo "Pid is $p"
                        pids+=($p)
                        echo "Start node $nodes"
                done
                for ((nodes = 0; nodes<$c; nodes++ ))
                do
                        echo "Wacht for pid ${pids[$nodes]}"
                        wait ${pids[$nodes]}
                done
                tijd1=`date +%s%3N`
                diff=`expr $tijd1 - $tijd`
                echo "$c;$verschil" >> results.csv
                echo "Stop time = $tijd1 start = $tijd time
                      taken = $diff"

        done
done
```

In table 1 are the results show of the different measurements. As earlier described the hypothesis is that adding more nodes will not linearly decrease the time that the system takes to be tested. From 1 to 2 nodes the increase is really big. Running this specific SUT this is also visible in the results. The time taken keeps decreasing until around 5 nodes. At 6 nodes there is almost no speed improvement anymore. The outcomes of the total wait time for the completion of all the nodes of TESTAR are registered here. The amount of states is checked from the database. Then the average of all the runs is taken in the column avg time taken and the factor is the speed improvement of the current time compared to the previous amount of nodes.

As seen in figure 9 it is clearly visible that adding nodes to the SUT makes improves the time it takes to complete all the different actions in the SUT. The relation between the amount of nodes and the time it takes is not linear. The vertical axis contains the time in seconds; the horizontal axis is the amount of nodes. When looking at figure 10 we see that in the end barely any improvement of time is seen. Especially going from five to six nodes does not make much sense anymore on this SUT. According to table 1 the improvement is

# of nodes	# of runs	# states	# edges	avg Time taken	factor	deviation of avg (seconds)
1	7	10	131	31m50	100 %	239,9
2	10	10	131	16m43	50 %	68,7
3	10	10	131	11m45	12 %	45,8
4	10	10	131	9m53	12,5 %	53,3
5	10	10	131	8m23	15 %	24,3
6	10	10	131	8m4	3 %	29,84

Table 1: Test results of TESTAR running on multiple instances using a shared state model

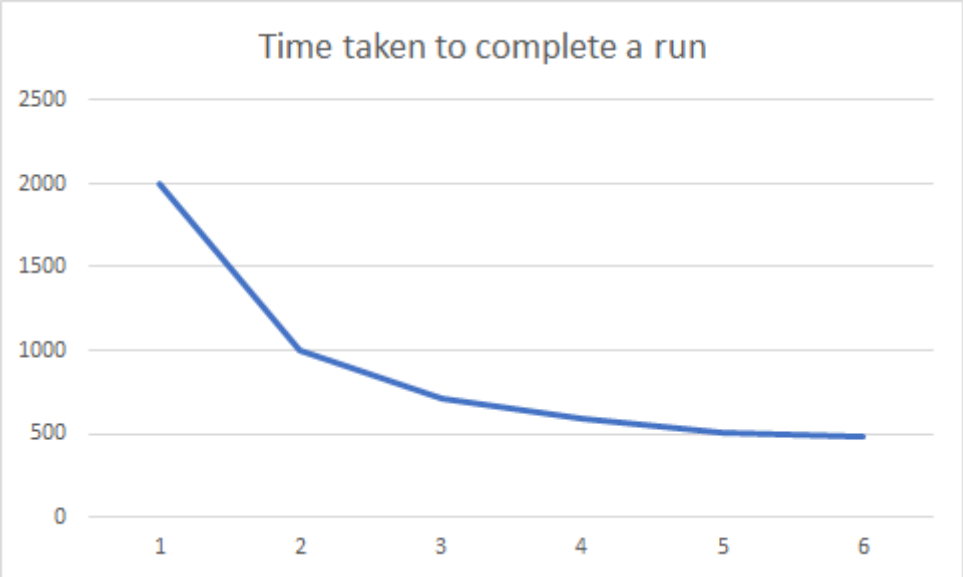

Figure 9: Time taken by the different nodes to complete the SUT

only three percent anymore. In this case the author comes to the conclusion that for the SUT that was used in these experiments, the boundary of the nodes to be used is around 5. Having 6 nodes does not improve the result anymore using the used configuration. However, it is possible that having another setup, for example, more memory available in the machine, placing the infrastructure on different locations, having a local version of the SUT instead of testing this SUT on the internet, would have impact.

7. Conclusion and Future Work

The research conducted in this thesis was about integrating TESTAR in an Azure DevOps environment. To be able to do so several things needed to be addressed. Those where described in the different research questions.

There where three research questions. The first question is about how to containerize TESTAR and create a shared state model so the different instances could work together. An algorithm is designed that follows the shortest paths to the unvisited actions. This algorithm is implemented and tested during this thesis. The second question was about integrating TESTAR in a build pipeline and being able to test the SUT. For this form filling and logging are implemented. The last research question is about what the performance gain is by using TESTAR in a parallel. For this, an experiment from 1 to 6 nodes was performed

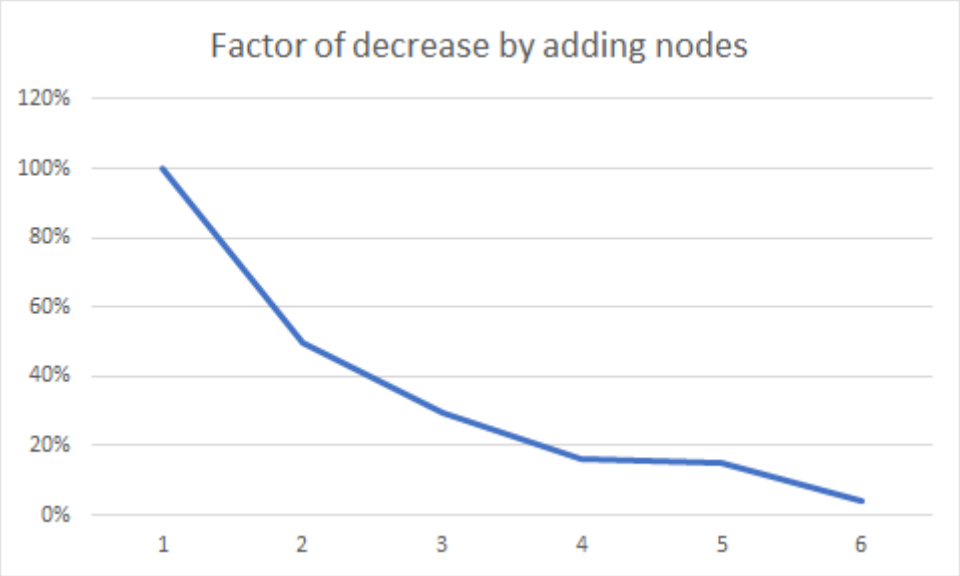

Figure 10: Time reduction in % by adding nodes

and speed improvements were measured using a statistical method.

In a DevOps environment releases take often place. Since TESTAR is changed during this thesis and major speed improvements are gained, TESTAR can be used in the company. Due to the infrastructural issues from the SUT this has not yet taken place, but this certainly will take place. Further research will be done on this topic and there are plans for writing a publication about it as well. The last part of this thesis had to do with being able to complete the tests. The company where the software has to be integrated has several registration processes that needed to be completed before even the normal sequence of actions can take place. The employers that register on the employer registration site have to complete three different steps. The employees that register on the employee portal even have to complete around six pages before the registration is complete and the normal process can start. Without the ability to fill the forms at once the testing software will probably never go to the final state where a user is registered. In this study, all the parts were realized, and even though not fully in use yet, it looks promising.

The experiments of this study were performed on the SUT Parabank. The expectation is that other SUTs will have the same line in the factor of time decrease. However, every SUT needs probably a different optimal amount of pods. The best results will be gathered during testing the SUT. The amount of nodes needed is strongly dependent on the amount of states in the model. The author recommends to start with around five pods in the beginning. This way an exploration of the GUI can be made, but significantly faster than doing this using one node. Using a single pod will take the maximum amount of time needed for the SUT. As soon as the pods get opportunities to select different paths this will be done uncovering the model faster.

To continue the work there are several parts the author thinks that are nice additions or extensions.

The NUNIT specification allows for storage and recording of screenshots. At this moment this is not implemented yet, however, it would be a nice feature. If following the logs through an test examination tool like Azure DevOps it would be possible to see the screen-

shots as well.

It would be possible to make the OrientDB part of the deployment. During the tests this was not needed since the author needed access to the OrientDB. In a normal situation this OrientDB could be deployed as well in the Kubernetes infrastructure file so it will be torn down after completing the test.

When a lot of nodes are setup in the first place, there are possibly more nodes than available actions. An algorithm could be devised so that the nodes will make a balanced decision to go as well to the states having the most available actions to execute.

A lot of work has been put in the ability to get TESTAR running on the WebDriver protocol. The same could be done as well for other browsers and other platforms. Selenium provides the community with selenium standalone packages for Chrome, Firefox, Opera and several others. Docker images for these browsers could be made as well. Appium is a platform for testing native mobile apps. Images for these could be made as well. Another interesting test platform could be the creation of Windows docker images. These are images running in Docker but based on the Windows platform. The accessibility API is then available so all the Windows applications can be tested as well.

When filling forms, validation can take place on all kinds of script injections to see whether the tested website is prone to security bugs as well. When performing script injection, tests can be done whether for example a popup window is being shown or certain text in the page is modified.

There are nice possibilities to enhance the form filling methodology. For example at this moment, forms are stored on disk with predefined values. An alternative would be to let the TESTAR guess what values can be used randomly. When a form can complete with the random values add those as well to the file and make a random action selection of the forms available for this specific action. Finally complete with a form is certainly accepted so a next step can be taken. An alternative as well would be to define regular expressions with values that are allowed. This however limits the testing abilities but will extend the amount of possible inputs in the fields.

8. APPENDICES

8.1. APPENDIX A

```
apiVersion: v1
kind: Service
metadata:
 name: orientservice
spec:
  selector:
   app: orientservice
   tier: orient
  ports:
  - name: porta
    port: 2480
    targetPort: http
  - name: portb
    port: 2424
    targetPort: http2
```

```
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: orientservice
spec:
 replicas: 0
 selector:
   matchLabels:
    app: orientservice
    tier: orient
    track: stable
 template:
   metadata:
     labels:
      app: orientservice
      tier: orient
      track: stable
   spec:
     containers:
     - name: orientservice
       image: orientdb:3.0.34
       ports:
       - name: http
         containerPort: 2480
       - name: http2
         containerPort: 2424
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: settings
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.152.187.5
    path: "/export/testar/settings"
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: output
spec:
  capacity:
    storage: 1Mi
```

```
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.152.187.5
    path: "/export/testar/output"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: settings
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  volumeName: settings
  resources:
    requests:
      storage: 1Mi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: output
spec:
  accessModes:
    - ReadWriteMany
  volumeName: output
  storageClassName: ""
  resources:
    requests:
      storage: 1Mi
---
apiVersion: batch/v1
kind: Job
metadata:
  name: testar
spec:
  ttlSecondsAfterFinished: 100
  parallelism: 5
  template:
    metadata:
      labels:
        app: testar
    spec:
      restartPolicy: Never
      containers:
      - name: testar
        image: aslomp/testar
        volumeMounts:
```

```
        - mountPath: /mnt
          name: config
        - mountPath: /testar/bin/settings
          name: config
        - mountPath: /testar/bin/output
          name: logs
        - mountPath: /dev/shm
          name: dshm
      volumes:
      - name: config
        persistentVolumeClaim:
          claimName: settings
      - name: logs
        persistentVolumeClaim:
          claimName: output
      - name: dshm
        emptyDir:
          medium: Memory
```

8.2. APPENDIX B

```
/**
 *
 * Copyright (c) 2018, 2019 Open Universiteit - www.ou.nl
 * Copyright (c) 2019 Universitat Politecnica de Valencia - www.upv.es
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the copyright holder nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 */

import es.upv.staq.testar.NativeLinker;
import es.upv.staq.testar.protocols.ClickFilterLayerProtocol;

import org.fruit.Pair;
import org.fruit.alayer.*;
import org.fruit.alayer.actions.*;
import org.fruit.alayer.exceptions.ActionBuildException;
import org.fruit.alayer.exceptions.NoSuchTagException;
import org.fruit.alayer.exceptions.StateBuildException;
import org.fruit.alayer.exceptions.SystemStartException;
import org.fruit.alayer.webdriver.*;
import org.fruit.alayer.webdriver.enums.WdRoles;
import org.fruit.alayer.webdriver.enums.WdTags;
import org.fruit.monkey.ConfigTags;
import org.fruit.monkey.Settings;
import org.testar.protocols.DesktopProtocol;
import org.testar.protocols.WebdriverProtocol;
import org.fruit.alayer.webdriver.WdProtocolUtil;
import nl.ou.testar.StateModel.Persistence.*;
import nl.ou.testar.StateModel.*;
import nl.ou.testar.StateModel.Persistence.OrientDB.*;
import nl.ou.testar.StateModel.Persistence.OrientDB.Entity.Config;
```

```
import nl.ou.testar.StateModel.Persistence.OrientDB.Entity.Connection;
import nl.ou.testar.StateModel.Persistence.OrientDB.Entity.EntityManager;

import java.util.*;
import java.lang.Thread;
import java.net.*;

import org.w3c.dom.*;

import javax.swing.text.html.Option;
import javax.xml.parsers.*;
import java.io.*;

import static org.fruit.alayer.Tags.Blocked;
import static org.fruit.alayer.Tags.Enabled;
import static org.fruit.alayer.webdriver.Constants.scrollArrowSize;
import static org.fruit.alayer.webdriver.Constants.scrollThick;
import nl.ou.testar.StateModel.Persistence.*;
import org.fruit.alayer.actions.CompoundAction.*;
import com.orientechnologies.orient.core.db.ODatabaseSession;
import com.orientechnologies.orient.core.exception.OConcurrentModificationException;
import com.orientechnologies.orient.core.id.ORecordId;
import com.orientechnologies.orient.core.db.*;
import com.orientechnologies.orient.core.sql.executor.*;
import com.orientechnologies.orient.core.record.*;

public class Protocol_webdriver_unvisited extends WebdriverProtocol {
        // Classes that are deemed clickable by the web framework

        OrientDBManager odb;
        ModelManager m;
        String nodeName = "";
        EntityManager em;
        String selectedAction = null;
        OrientDB database;

        Settings settings;

        public Protocol_webdriver_unvisited() {

        }


        private static Pair<String, String> login = null;
        private static Pair<String, String> username = Pair.from("username", "");
        private static Pair<String, String> password = Pair.from("password", "");

        // List of atributes to identify and close policy popups
        // Set to null to disable this feature
        private static Map<String, String> policyAttributes = new HashMap<String, String>() {
                {
                        put("id", "sncmp-banner-btn-agree");
                }
        };


        ODatabaseSession CreateDatabaseConnection() {

                // connection = new Connection(database,
                // OrientDBManagerFactory.getDatabaseConfig(settings));
                Config config = OrientDBManagerFactory.getDatabaseConfig(settings);
                ODatabaseSession dbSession = database.open(config.getDatabase(), config.getUser(), config.getPassword());
                System.out.println("Own database connection created");
                return dbSession;
        }

        @Override
        protected void beginSequence(SUT system, State state) {
                super.beginSequence(system, state);
                System.out.println("BeginSequence");
                _moreActions = true;
                stopSequences = false;
        }

        boolean hasHadActionsInDb = false;

        boolean stopSequences = false;

        @Override
        protected void initialize(Settings settings) {
                NativeLinker.addWdDriverOS();
                deniedExtensions = Arrays.asList("pdf", "jpg", "png", "wsdl", "pfx");

                clickableClasses = Arrays.asList("v-menubar-menuitem", "v-menubar-menuitem-caption");
                super.initialize(settings);
                // domainsAllowed = Arrays.asList("https://para.testar.org");
                ensureDomainsAllowed();
                this.settings = settings;
                // Propagate followLinks setting
                WdDriver.followLinks = true;


                m = (ModelManager) stateModelManager;
                if (m.persistenceManager != null) {
                        System.out.println("PersistenceManager != null type = " + m.persistenceManager.getClass());
```

```
                odb = (OrientDBManager) m.persistenceManager;
                em = odb.entityManager;
                database = new OrientDB(EntityManager.getConnectionString(), OrientDBConfig.defaultConfig());

                System.out.println("dbSession initialized");
                Random r = new Random();
                nodeName = System.getenv("HOSTNAME") + "_" + r.nextInt(10000);
                System.out.println("nodeName = " + nodeName);
                ODatabaseSession dbSession = CreateDatabaseConnection();
                ExecuteCommand(dbSession, "create vertex BeingExecuted set node = '" + nodeName + "'").close();
                dbSession.close();

        }

}

public OResultSet ExecuteCommand(ODatabaseSession db, String actie) {
        System.out.println("ExecuteCommand " + actie);
        boolean repeat = false;
        do {
                repeat = false;
                try {
                        System.out.println("dbSession = " + db);
                        return db.command(actie);
                } catch (OConcurrentModificationException ex) {
                        repeat = true;
                        try {
                                Thread.sleep(2000);
                        } catch (Exception e) {
                        }

                }
        } while (repeat);
        return null;

}

public OResultSet ExecuteQuery(ODatabaseSession db, String actie) {
        System.out.println("ExecuteQuery " + actie);
        boolean repeat = false;
        do {
                repeat = false;
                try {
                        System.out.println("dbSession = " + db);

                        return db.query(actie);
                } catch (OConcurrentModificationException ex) {
                        repeat = true;
                        try {
                                Thread.sleep(2000);
                        } catch (Exception e) {
                        }

                }
        } while (repeat);

        return null;

}

/**
 * This method is called when TESTAR starts the System Under Test (SUT). The
 * method should take care of 1) starting the SUT (you can use TESTAR's settings
 * obtainable from <code>settings()</code> to find out what executable to run)
 * 2) bringing the system into a specific start state which is identical on each
 * start (e.g. one has to delete or restore the SUT's configuratio files etc.)
 * 3) waiting until the system is fully loaded and ready to be tested (with
 * large systems, you might have to wait several seconds until they have
 * finished loading)
 *
 * @return a started SUT, ready to be tested.
 */
@Override
protected SUT startSystem() throws SystemStartException {
        return super.startSystem();
}

/**
 * This method is used by TESTAR to determine the set of currently available
 * actions. You can use the SUT's current state, analyze the widgets and their
 * properties to create a set of sensible actions, such as: "Click every Button
 * which is enabled" etc. The return value is supposed to be non-null. If the
 * returned set is empty, TESTAR will stop generation of the current action and
 * continue with the next one.
 *
 * @param system the SUT
 * @param state  the SUT's current state
 * @return a set of actions
 */

/*
 * public CompoundAction fillForm(HashMap<org.fruit.alayer.Widget, String>
 * values) { StdActionCompiler ac = new AnnotatingActionCompiler();
 *
 * CompoundAction a = CompoundAction.Builder(); values.forEach((k,v)->
```

```
 * a.add(ac.clickTypeInto(k, v, true)));
 *
 * return a; }
 */
@Override
protected Set<Action> deriveActions(SUT system, State state) throws ActionBuildException {
        // Kill unwanted processes, force SUT to foreground
        System.out.println("Protocol file: deriveActions: Arend acties");
        Set<Action> actions = super.deriveActions(system, state);

        // create an action compiler, which helps us create actions
        // such as clicks, drag&drop, typing ...
        StdActionCompiler ac = new AnnotatingActionCompiler();

        // Check if forced actions are needed to stay within allowed domains
        if (WdDriver.getCurrentUrl().contains("wsdl") || WdDriver.getCurrentUrl().contains("wadl")) {
                return new HashSet<>(Collections.singletonList(new WdHistoryBackAction()));

        }
        Set<Action> forcedActions = detectForcedActions(state, ac);
        if (forcedActions != null && forcedActions.size() > 0) {
                System.out.println("Executing forced action");
                return forcedActions;
        }

        // iterate through all widgets
        for (org.fruit.alayer.Widget widget : state) {
                // only consider enabled and non-tabu widgets

                // System.out.println("DeriveAction widget = "+widget+" class =
                // "+widget.getClass());
                WdWidget wd = (WdWidget) widget;
                WdElement element = wd.element;
                // System.out.println("DeriveAction widget = "+widget+" class =
                // "+widget.getClass()+" wd = "+wd+" elem= "+element.tagName);
                if (isForm(widget))
                {
                 System.out.println("Form gevonden");
                 fillForm(actions, ac, state, wd, null);

                }
                if (!widget.get(Enabled, true) || blackListed(widget)) {
                        continue;
                }

                // slides can happen, even though the widget might be blocked

                // If the element is blocked, Testar can't click on or type in the widget
                if (widget.get(Blocked, false)) {
                        continue;
                }

                // type into text boxes; Replaced by Form Filling algorithm
                // if (isAtBrowserCanvas(widget) && isTypeable(widget) && (whiteListed(widget)
                // || isUnfiltered(widget))) {
                // actions.add(ac.clickTypeInto(widget, this.getRandomText(widget), true));
                // }

                // left clicks, but ignore links outside domain
                if (isAtBrowserCanvas(widget) && isClickable(widget) && (whiteListed(widget) || isUnfiltered(widget))) {
                        if (!isLinkDenied(widget) && !mijnIgnore(widget)) {
                                actions.add(ac.leftClickAt(widget));
                        }
                }
        }
        if (actions.size() == 0) {
                System.out.println("Actions.size == 0; Return historyback and done with DeriveActions");
                return new HashSet<>(Collections.singletonList(new WdHistoryBackAction()));
        }
        System.out.println("Done with DeriveActions");

        return actions;
}

public boolean mijnIgnore(org.fruit.alayer.Widget widget) {
        String linkUrl = widget.get(Tags.ValuePattern, "");

        if (linkUrl.contains("wsdl") || linkUrl.contains("parasoft.com") || linkUrl.contains("/services/")
                        || linkUrl.contains("api-docs") || linkUrl == "" || linkUrl.contains(".pfx") || linkUrl.contains("wadl")
                        || linkUrl.contains("xml")) {
                System.out.println("negeer " + linkUrl);
                return true;
        }
        System.out.println("accepteer " + linkUrl);
        return false;
}

public HashMap<String, String> readFormFile(String fileName) {
        try {
                DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
                DocumentBuilder builder = factory.newDocumentBuilder();
                Document document = builder.parse(new File(fileName));
                document.getDocumentElement().normalize();
                Element root = document.getDocumentElement();
                System.out.println("readFormFile");
```

```
                NodeList items = root.getChildNodes();
                HashMap<String, String> result = new HashMap<String, String>();
                for (int i = 0; i < items.getLength(); i++) {

                        Node item = items.item(i);
                        Element node = (Element) item;
                        String value = node.getTextContent();
                        // System.out.println(node.getNodeName() +"=" +value);
                        result.put(node.getNodeName(), value);
                }
                System.out.println("Einde formfile");
                return result;
        } catch (Exception e) {
        }
        return null;

}

public void storeToFile(String fileName, HashMap<String, String> data) {
        String result = "<form><performSubmit>true</performSubmit>";

        for (Map.Entry<String, String> entry : data.entrySet()) {
                String key = entry.getKey();
                String value = entry.getValue();

                result += "<" + key + ">" + value + "</" + key + ">";
        }
        result += "</form>";
        try {
                BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
                writer.write(result);

                writer.close();
        } catch (Exception e) {
        }
        // System.out.println(result);

}

boolean inForm = false;

@Override
protected boolean blackListed(org.fruit.alayer.Widget w) {
        if (inForm)
                return false;

        return super.blackListed(w);
}

public int buildForm(CompoundAction.Builder caB, WdWidget widget, HashMap<String, String> fields, boolean storeFile,
                StdActionCompiler ac) {
        int sum = 0;
        WdElement element = widget.element;
        String defaultValue = "write-random-genenerated-value";
        if (isTypeable(widget)) {

                if (storeFile) {

                        fields.put(element.name, defaultValue);
                }
                if (fields.containsKey(element.name) && fields.get(element.name) != null) {
                        caB.add(ac.clickTypeInto(widget, fields.get(element.name), true), 2);
                        sum += 2;
                }

        }
        String baseElem = element.tagName;
        // System.out.println("check children: huidig element "+element.tagName+" aantal
        // childs: "+widget.childCount());
        for (int i = 0; i < widget.childCount(); i++) {
                WdWidget w = widget.child(i);
                // System.out.println("child "+i+" van element "+baseElem);
                // WdElement
                element = w.element;

                if (isTypeable(w)) {

                        if (storeFile) {
                                fields.put(element.name, defaultValue);
                        }
                        if (fields.containsKey(element.name) && fields.get(element.name) != null) {
                                caB.add(ac.clickTypeInto(widget, fields.get(element.name), true), 2);
                                sum += 2;
                        }

                } else {
                        sum += buildForm(caB, widget.child(i), fields, storeFile, ac);
                }

        }
        return sum;

}

public void fillForm(Set<Action> actions, StdActionCompiler ac, State state, WdWidget widget,
```

```
                HashMap<String, String> fields) {
        // System.out.println("Url = "+WdDriver.getCurrentUrl());
        inForm = true;
        if (fields == null) {
                fields = new HashMap<String, String>();
        }
        URI uri = null;
        try {
                uri = new URI(WdDriver.getCurrentUrl());
        } catch (Exception e) {
        }
        String formId = widget.getAttribute("name");
        if (formId == null) {
                formId = "";
        }
        String path = (uri.getPath() + "/" + formId).replace("/", "_") + ".xml";
        System.out.println("Look for file " + path);
        File f = new File(path);
        Boolean storeFile = true;
        if (f.exists()) {
                storeFile = false;
                fields = readFormFile(path);
                System.out.println("Bestand bestaat, lees de data uit bestand");
        }
        CompoundAction.Builder caB = new CompoundAction.Builder();
        int sum = buildForm(caB, widget, fields, storeFile, ac);

        if (fields.containsKey("performSubmit")) {
                boolean submit = Boolean.getBoolean(fields.get("performSubmit"));
                if (submit && formId != "") {

                        caB.add(new WdSubmitAction(formId), 2);
                }
        }
        if (storeFile) {
                storeToFile(path, fields);
        }
        if (sum > 0) {
                CompoundAction ca = caB.build();

                actions.add(ca);
        }
        inForm = false;
        System.out.println("fillForm klaar");
        // return ca;
}

boolean _moreActions = true;

@Override
protected boolean moreActions(State state) {
        System.out.println("MoreActions: _moreActions = " + _moreActions);
        return _moreActions;
}

boolean stop = false;

@Override
protected boolean moreSequences() {
        boolean result = (CountInDb("unvisitedabstractaction") > 0) || !stop;

        System.out.println("moreSequences: " + result);
        return result;
}

@Override
protected boolean isClickable(org.fruit.alayer.Widget widget) {
        Role role = widget.get(Tags.Role, Roles.Widget);
        if (Role.isOneOf(role, NativeLinker.getNativeClickableRoles())) {
                // Input type are special...
                if (role.equals(WdRoles.WdINPUT)) {
                        String type = ((WdWidget) widget).element.type;
                        return WdRoles.clickableInputTypes().contains(type);
                }
                return true;
        }

        WdElement element = ((WdWidget) widget).element;
        if (element.isClickable) {
                return true;
        }

        Set<String> clickSet = new HashSet<>(clickableClasses);
        clickSet.retainAll(element.cssClasses);
        return clickSet.size() > 0;
}

boolean isForm(org.fruit.alayer.Widget widget) {
        Role r = widget.get(Tags.Role, Roles.Widget);
        if (Role.isOneOf(r, new Role[] { WdRoles.WdFORM })) {
                return r.equals(WdRoles.WdFORM);
        }
        return false;
}
```

```
@Override
protected boolean isTypeable(org.fruit.alayer.Widget widget) {
        Role role = widget.get(Tags.Role, Roles.Widget);
        if (Role.isOneOf(role, NativeLinker.getNativeTypeableRoles())) {
                // Input type are special...
                if (role.equals(WdRoles.WdINPUT)) {

                        String type = ((WdWidget) widget).element.type.toLowerCase();
                        return WdRoles.typeableInputTypes().contains(type);
                }
                return true;
        }
        return false;
}

/**
 * Select one of the available actions using an action selection algorithm (for
 * example random action selection)
 *
 * @param state   the SUT's current state
 * @param actions the set of derived actions
 * @return the selected action (non-null!)
 */

private Boolean finishedAction = false;

public ArrayList<String> GetUnvisitedActionsFromDatabase(String currentAbstractState) {
        System.out.println("GetUnvisitedActionsFromDatabase");
        ArrayList<String> result = new ArrayList<String>();
        String sql = "SELECT expand(path) FROM (  SELECT shortestPath($from, $to) AS path   LET     $from = (SELECT FROM abstractstate WHERE stateId='"
                        + currentAbstractState + "'),     $to = (SELECT FROM BlackHole)   UNWIND path)";
        System.out.println(sql);
        OResultSet rs = null;
        ODatabaseSession db = CreateDatabaseConnection();
        try {
                rs = ExecuteQuery(db, sql);

                while (rs.hasNext()) {
                        OResult item = rs.next();
                        if (item.isVertex()) {
                                System.out.println("Item is a vertex");
                                Optional<OVertex> optionalVertex = item.getVertex();

                                OVertex nodeVertex = optionalVertex.get();
                                for (OEdge edge : nodeVertex.getEdges(ODirection.OUT, "UnvisitedAbstractAction")) {
                                        result.add(edge.getProperty("actionId"));
                                        System.out.println("Edge " + edge + " gevonden ID = " + edge.getProperty("actionId"));
                                }

                        }

                        System.out.println("friend: " + item);
                }

        } catch (Exception e) {
                System.out.println("Exception during GetUnvisitedActionsFromDatabase " + e);
                e.printStackTrace();

        } finally {
                rs.close();
                db.close();

        }
        System.out.println("Klaar met ophalen acties");
        return result;
}

public long CountInDb(String table) {
        long aantal = 0;
        String sql = "SELECT count(*) as aantal from " + table;
        ODatabaseSession db = CreateDatabaseConnection();
        try {
                OResultSet rs = ExecuteQuery(db, sql);
                OResult item = rs.next();
                aantal = item.getProperty("aantal");
                rs.close();
                System.out.println(sql + "  aantal =  " + aantal);
        } finally {

                db.close();
        }
        if (aantal > 0) {
                hasHadActionsInDb = true;
        }

        return aantal;

}

public void UpdateAbstractActionInProgress(String actionId) {
        ODatabaseSession db = CreateDatabaseConnection();
        try {
                System.out.println("actie  AbstractID = " + actionId);

                String sql = "update edge UnvisitedAbstractAction set in = (SELECT FROM BeingExecuted WHERE node='"
```

```
                        + nodeName + "') where actionId='" + actionId + "'";
                System.out.println("Execute" + sql);

                ExecuteCommand(db, sql);

        } catch (Exception e) {
                System.out.println("Can not update unvisitedAbstractAction; set selectedAction to null");
                selectedAction = null;
        } finally {
                db.close();
        }

}

public HashMap<String, Action> ConvertActionSetToDictionary(Set<Action> actions) {
        System.out.println("Convert Set<Action> to HashMap containing actionIds as keys");
        HashMap<String, Action> actionMap = new HashMap<String, Action>();
        ArrayList<Action> actionList = new ArrayList<Action>(actions);
        for (Action a : actionList) {
                System.out.println(
                        "Add action " + a.get(Tags.AbstractIDCustom) + "  to actionMap; description = " + a.get(Tags.Desc));
                actionMap.put(a.get(Tags.AbstractIDCustom), a);
        }
        System.out.println("ActionMap initialized");
        return actionMap;
}

public String selectRandomAction(ArrayList<String> actions) {

        long graphTime = System.currentTimeMillis();
        Random rnd = new Random(graphTime);

        String ac = actions.get(rnd.nextInt(actions.size()));
        UpdateAbstractActionInProgress(ac);
        System.out.println("Update action " + ac + " from BlackHole to BeingExecuted");
        return ac;
}

int cyclesWaitBeforeNewAction = 0;

String lastState = "";
int sameState = 0;

private String getNewSelectedAction(State state, Set<Action> actions) {
        String result = null;
        System.out.println("getNewSelectedAction state = " + state.get(Tags.AbstractIDCustom));
        Boolean ok = false;
        do {
                try {
                        ArrayList<String> availableActionsFromDb = GetUnvisitedActionsFromDatabase(
                                        state.get(Tags.AbstractIDCustom));

                        System.out.println(
                                        "Number of shortest path actions available in database: " + availableActionsFromDb.size());

                        if (availableActionsFromDb.size() >= 1) {
                                ok = true;
                                String action = selectRandomAction(availableActionsFromDb);
                                System.out.println("Action from database selected: action = " + action);
                                return action;
                        }

                        long aantalAbstractActions = CountInDb("abstractstate");

                        System.out.println(
                                        "No actions available in database; abstract states in database = " + aantalAbstractActions);

                        if (aantalAbstractActions == 0) {
                                Action a = super.selectAction(state, actions);
                                String action = a.get(Tags.AbstractIDCustom);
                                System.out.println(
                                                "Since this is the first run and no abstractactions exists take a random action; statemodel is probably laggi
                                                        + action);
                                return action;
                        }

                        // System.out.println("Not allowed to be here! or really done");
                        _moreActions = false;
                        stop = true;
                        Action a = super.selectAction(state, actions);
                        String action = a.get(Tags.AbstractIDCustom);
                        System.out.println("Just return random action and stop; action = " + action);
                        return action;

                } catch (Exception e) {
                        int sleepTime = new Random(System.currentTimeMillis()).nextInt(5000);
                        System.out.println("Exception while getting action; Wait " + sleepTime + " ms " + e);

                        ok = false;
                        try {
                                Thread.sleep(sleepTime);
                        } catch (Exception th) {

                        }
                }
```

```
        } while (!ok);
        return result;
    }

    class TmpData {
        public TmpData(ORecordId rid, String stateId) {
            this.stateId = stateId;
            this.rid = rid.toString();
            System.out.println("TmpData " + stateId + "  " + rid);
        }

        public String stateId;
        public String rid;
    }

    public void ReturnActionToBlackHole() {
        ODatabaseSession db = CreateDatabaseConnection();
        try {
            String sql = "update unvisitedabstractaction set in = (select from blackhole) where actionId='"
                    + selectedAction + "'";
            System.out.println("Return action to blackhole from beingexecuted: " + selectedAction + " sql = " + sql);
            ExecuteCommand(db, sql);
        } catch (Exception e) {
            System.out.println("Not possible to return selectedAction " + selectedAction + "  to blackhole" + e);
        } finally {
            db.close();
        }
    }

    public Action traversePath(State state, Set<Action> actions) {

        String q1 = "select stateId from abstractstate where @rid in (select outV() from UnvisitedAbstractAction where actionId = '"
                + selectedAction + "')";

        String destinationStateId = "";
        ODatabaseSession db = CreateDatabaseConnection();

        OResultSet destinationStatResultSet = ExecuteQuery(db, q1);
        if (destinationStatResultSet.hasNext()) {
            OResult item = destinationStatResultSet.next();
            System.out.println("destinationResultSet item = " + item);
            // Optional<OVertex> optionalVertex = item.getProperty("stateId");
            destinationStateId = item.getProperty("stateId");
            System.out.println("traversePath: onderweg naar " + destinationStateId);
            destinationStatResultSet.close();
            db.close();
        } else {
            System.out.println(
                    "State die vastzit aan de unvisitedaction niet gevonden; zet selectedAction op null; voer nu historyback uit");
            destinationStatResultSet.close();
            ReturnActionToBlackHole();
            selectedAction = null;
            return new WdHistoryBackAction();
        }

        // haal stateId op van q1

        // SELECT @rid, stateId from (
        // SELECT expand(path) FROM ( SELECT shortestPath($from,
        // $to,'OUT','AbstractAction') AS path LET $from = (SELECT FROM abstractstate
        // WHERE stateId='SAC1jp4oysed31697927673'), $to = (SELECT FROM abstractstate
        // Where stateId='SACwpszr27b61710690312') UNWIND path))
        String q2 = "SELECT @rid, stateId from (SELECT expand(path) FROM (  SELECT shortestPath($from, $to,'OUT','AbstractAction') AS path
LET     $from = (SELECT FROM abstractstate WHERE stateId='"
                    + state.get(Tags.AbstractIDCustom) + "'),      $to = (SELECT FROM abstractstate Where stateId='"
                    + destinationStateId + "')   UNWIND path))";
        db = CreateDatabaseConnection();
        OResultSet pathResultSet = ExecuteQuery(db, q2);
        Vector<TmpData> v = new Vector<TmpData>();

        while (pathResultSet.hasNext()) {
            OResult item = pathResultSet.next();
            v.add(new TmpData(item.getProperty("@rid"), item.getProperty("stateId")));

        }
        pathResultSet.close();
        db.close();

        if (v.size() < 2) {
            System.out.println(
                    "Er bestaat geen pad! Uitvoeren super.selectAction; Eindig sequence ook door _moreActions = false te zetten");
            ReturnActionToBlackHole();
            selectedAction = null;
            _moreActions = false;
            return super.selectAction(state, actions);

            // Er bestaat geen pad
        }

        // Zoek uit te voeren abstractaction
        String q3 = "select from abstractaction where out = " + v.get(0).rid + " and in = " + v.get(1).rid;

        String abstractActionId = "";
        var beschikbareActions = ConvertActionSetToDictionary(actions);
```

```
            db = CreateDatabaseConnection();
            OResultSet abstractActionResultSet = ExecuteQuery(db, q3);
            while (abstractActionResultSet.hasNext()) {
                abstractActionId = abstractActionResultSet.next().getProperty("actionId");
                System.out.println("traversePath: gebruik hiervoor action " + abstractActionId);

                System.out.println("Check of " + abstractActionId + "  beschikbaar is");
                if (beschikbareActions.containsKey(abstractActionId)) {
                    System.out.println(
                            "Actie " + abstractActionId + "  is beschikbaar is beschikbareActions; deze wordt uitgevoerd ");
                    abstractActionResultSet.close();
                    db.close();
                    return beschikbareActions.get(abstractActionId);
                }
            }
            abstractActionResultSet.close();
            db.close();

            System.out.println("Action that needs to be made does not exist");
            ReturnActionToBlackHole();
            selectedAction = null;

            return super.selectAction(state, actions);

        }

        @Override
        protected Action selectAction(State state, Set<Action> actions) {

            // Call the preSelectAction method from the AbstractProtocol so that, if
            // necessary,
            // unwanted processes are killed and SUT is put into foreground.
            Action retAction = preSelectAction(state, actions);

            if (retAction != null) {
                return retAction;
            }

            // First check whether we do have a selected action; if not select one
            if (selectedAction == null) {

                selectedAction = getNewSelectedAction(state, actions);

            }

            if (selectedAction != null) {
                System.out.println("selectedAction != null actions.length = " + actions.size() + " state id = "
                        + state.get(Tags.AbstractIDCustom));
                HashMap<String, Action> actionMap = ConvertActionSetToDictionary(actions);
                System.out.println("actionMap.size() = " + actionMap.size());
                // select path towards the selected action
                if (actionMap.containsKey(selectedAction)) {
                    System.out.println("Needed action is currently available, so select this action");
                    // Perform desired action since it's available from this point
                    Action a = actionMap.get(selectedAction); // Get the AbstractAction matching the selectedAction
                    selectedAction = null; // Reset selectedAction so next time a new one will be choosen.
                    return a;
                } else {
                    System.out.println("Needed action is unavailable, select from path to be followed");
                    Action a = traversePath(state, actions);
                    return a;
                }
            }

            if (retAction != null)
                return retAction;
            System.out.println("Return a fallback action");

            retAction = super.selectAction(state, actions);

            return retAction;

        }

}
```

BIBLIOGRAPHY

[1] Pekka Aho, Tanja E. J. Vos, Sami Ahonen, Tomi Piirainen, Perttu Moilanen, and Fernando Pastor Ricos. Continuous piloting of an open source test automation tool in an industrial environment. 2020. 7

[2] Docker. What is a container? 2020. https://www.docker.com/resources/what-container. 5

[3] Kubernetes.io. Create an external load balancer. 2020. https:

//kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/. 6

[4] Kubernetes.io. Deployments | kubernetes. 2020. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. 6

[5] Kubernetes.io. Ingress | kuberenetes. 2020. https://kubernetes.io/docs/concepts/services-networking/ingress/. 6

[6] Kubernetes.io. Pods | kubernetes. 2020. https://kubernetes.io/docs/concepts/workloads/pods/. 6, 8

[7] Kubernetes.io. Services | kubernetes. 2020. https://kubernetes.io/docs/concepts/services-networking/service/. 6

[8] John A. Matthews. Distributed automated software graphical user interface (gui) testing. 2001. https://patentimages.storage.googleapis.com/db/45/b0/82323d70d05fe8/US7055137.pdf. 8

[9] Microsoft. Choosing the right version control for your project. 2020. https://docs.microsoft.com/en-us/azure/devops/repos/tfvc/comparison-git-tfvc?view=azure-devops. 6

[10] Ad Mulders. Infering state models in testar. 2020. 7, 8, 10

[11] The NUNIT Project. Specifications. 2021. https://docs.nunit.org/articles/nunit/technical-notes/nunit-internals/specs/Specifications.html. 20

[12] Fernando Pastor Ricós, Pekka Aho, Tanja Vos, Ismael Torres Boigues, Ernesto Calás Blasco, and Héctor Martínez Martínez. Deploying testar to enable remote testing in an industrial ci pipeline: A case-based evaluation. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 543–557, Cham, 2020. Springer International Publishing. 8

[13] Selenium. Selenium projects. 2021. https://www.selenium.dev/projects/. 5

[14] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014. 6

[15] Tanja Vos, Peter Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design*, 6:46–83, 07 2015. 7