Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: *Computer Science*	Spring semester, 2016 Open Access
Author: *Janicke Falch*	.... (signature author)
Instructor: *Hein Meling* External Supervisor: *Kristin Dahle Larsen (Altibox)*	
Title of Master's Thesis: *OptiRun: A Platform for Optimized Test Execution in Distributed Environments*	
ECTS: *30*	
Subject headings: *Test Automation · Automated Testing · Software Testing · Optimization · Algorithms · Allocation · Constraint Programming · Scheduling · Python · Selenium · Django*	Pages: 95 + Attachments + 2 x USB memory stick Stavanger, 15.06/2016 Date/year

OptiRun

A Platform for Optimized Test Execution in Distributed Environments

Janicke Falch — June 15, 2016

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

"Things were going badly; there was something wrong in one of the circuits of the long glass-enclosed computer. Finally, someone located the trouble spot and, using ordinary tweezers, removed the problem, a two-inch moth. From then on, when anything went wrong with a computer, we said it had bugs in it."

— Grace Hopper

Abstract

Computer systems have grown to play an essential role in our society. Software testing is an important asset of the software development process, as it serves to evaluate the quality of the software object being tested. Some software tests can be automated to obtain benefits such as increased efficiency, expanded test coverage and saved time.

This thesis presents *OptiRun*; a platform for optimized execution of automated tests in distributed systems. OptiRun consists of two main elements; a controller, which is responsible for managing the distributed system as well as for test allocation, execution and result reporting, and a web-based user interface where users can upload and manage test scripts, request immediate test executions and schedule executions ahead of time as well as report failed test executions to the issue tracking system *JIRA*. Tests on the user's level can be time-consuming, so the design and implementation of a test allocation mechanism named *OptiX* represented a major objective of this thesis. OptiX seeks to allocate tests to machines such that the overall execution time of test sets is attempted minimized. An alternative allocation mechanism was implemented for benchmarking purposes to better evaluate the performance of OptiX. An experimental evaluation and comparison of the two methods, using an extensive series of pseudo test sets, is also presented the thesis. This project is written in the *Python* programming language, and is built upon frameworks such as *Selenium* and *Django*.

OptiRun is intended to support the telecommunications company *Altibox* in the procedure of efficiently incorporating test automation as a practice in the testing process of their online web service *TV Overalt*.

Acknowledgements

First and foremost, I would like to express my sincere gratitude toward my instructor, Hein Meling, Professor at the Department of Electrical Engineering and Computer Science at the University of Stavanger, for his academic guidance and invaluable feedback throughout my work on this project. I would also like to thank Morten Mossige, Associate Professor at the Department of Electrical Engineering and Computer Science, for his contributions during the definition of the problem statement, as well as accessibility throughout the semester.

Further, I want to thank Kristin Dahle Larsen, Senior IPTV Engineer at Altibox, for her endless enthusiasm and for kindly accepting the task of being my external supervisor. Thanks are also due to IPTV Engineer Marius Eliassen, Head of IPTV Jarle Johnsen, and IPTV Test Manager Thomas Folgerø.

Last, but by no means least, I would like to offer my sincere thanks to my significant other, Samuel Trevena, for his helping hand in times of need, and for being an excellent resource for moral support and words of encouragement. Thank you for giving me the drive to finish this project!

Contents

1 Introduction **1**

1.1 Origin 1

1.2 Motivation 2

1.3 Purpose 2

1.4 Outline 3

2 Background **4**

2.1 Software Testing 4

2.1.1 The V-Model 5

2.1.2 Black-Box & White-Box Testing 6

2.1.3 Test Automation 6

2.2 Constraint Programming & Optimization 7

2.3 Related Work 8

2.3.1 TC-Sched 9

2.3.2 Sauce Labs 9

2.3.3 The Autograder Project 10

3 Technology **11**

3.1 The Python Programming Language 11

3.2 Selenium 11

3.2.1 Selenium WebDriver 12

3.2.2 Selenium Grid 13

3.3 Django 14

3.4 OR-Tools 14

4 System Overview **16**

4.1 Architecture 16

4.2 User Interface 18

4.2.1 Home Screen 18

4.2.2 Authentication & Authorization 18

4.2.3 Test Case Module 18

4.2.4 Test Group Module 20

4.2.5 Scheduling Module 20

4.2.6 Execution Log 21

4.2.7 Test Machines 22

4.2.8 Other 23

5 Design & Implementation **25**

5.1 Selenium Grid Integration 25

5.2 Controller 27

5.2.1 Selenium Server Listener 27

5.2.2 Test Machine Manager 28

5.2.3 Request Listener 29

5.2.4 Schedule Listener 29

5.2.5 Test Executor & Queue System 30

5.3 Database 32

5.4 Web-Based User Interface 33

5.4.1 Models 34

5.4.2 Admin 35

5.4.3 Issue Tracker Reporting 36

5.4.4 Event Recurrence 38

6 Test Allocation Mechanism **40**

6.1 OptiX 40

6.1.1 Preliminary Sorting 41

6.1.2 Initial Allocation 42

6.1.3 Improvement Iterations 44
6.2 ORX 48

7 Evaluation 50
7.1 Experimental Evaluation 50
7.1.1 Test Data 50
7.1.2 Test Data Collection 1 54
7.1.3 Test Data Collection 2 55
7.1.4 Test Data Collection 3 56
7.1.5 Test Data Collection 4 57
7.1.6 Conclusion of Experimental Evaluation 57
7.2 Threats to Validity 58
7.3 Discussion 59
7.4 Return on Investment 61

8 Further Work 64
8.1 Extended Browser & Platform Support 64
8.2 App Testing 65
8.3 Notifications 65
8.4 Continuous Integration 66

9 Conclusion 67

Appendices 69

Appendix A Attachments 70
A.1 Program Files 70
A.2 Printer-Friendly Version of Thesis Report 70

Appendix B Setup Instructions 71
B.1 Windows 71
B.1.1 Server 72
B.1.2 Test Machine 73
B.2 Linux Test Machine 73

B.3 OR-Tools 74

Appendix C User Manual **75**

C.1 Writing and Uploading Test Scripts 75
C.2 Executing Tests 75
C.3 Scheduling Tests 76
C.4 Managing Test Machines 76
C.5 Test Results 76
C.6 User Administration 77
C.7 Troubleshooting Test Machines 77

Appendix D Poster **79**

1 Introduction

Software systems, covering enterprise products, commercial applications and everything in between, have grown to form a fundamental position in our society. People are becoming increasingly dependent on computers, and wish to be in control of the digital content they consume. As a consequence of digital content becoming progressively accessible, the importance of, and demand for, high-quality software has become substantial. Enhanced pressure on software vendors to deliver frequent releases of high-quality software requires efficiency in every stage of the software development process.

Software testing constitutes a central aspect of the software development process. It plays a critical role in quality assurance and defect detection, and is important in order to ensure that the software in question behaves as expected and according to specifications.

1.1 Origin

Altibox, a subsidiary of the Lyse Group, is a telecommunication company. They offer a selection of services that includes broadband, *Internet Protocol Television* (IPTV) and *Voice over Internet Protocol*, (VoIP). Since its origin, the company has grown significantly, and at a considerably higher pace than originally anticipated. Because of the rapid growth, the company neglected to apply a number of established business practices in several different areas, including the field of software testing.

IPTV has long been among Altibox' most prominent services, and has previously been available primarily through the use of TV decoders. In the later years, however, an important focus area of Altibox' has been to expand the availability of their TV and *Video On Demand* (VOD) content by applying *Over-The-Top* (OTT) technology, which means that the content is made accessible over the Internet. This is accomplished through the development of *TV Overalt*, which is available both as a web application for desktops and as mobile applications for mobile devices.

1.2 Motivation

Altibox currently perform all software testing manually. For quite some time, Altibox has wanted to automate some of the tests that are conducted manually on the web-based version of TV Overalt, but have failed to make it a priority. Since user-level tests generally are time consuming, Altibox needed a system that would let them execute such tests efficiently. They also wanted the tests to be executed in a controlled environment.

This thesis presents OptiRun: a tool that will help Altibox incorporate test automation according to their needs. The tool is not limited to TV Overalt, however, as it can be applied as a test platform for a broad range of web applications.

1.3 Purpose

The aim of this project was to design and create a tool that would substantiate test automation with the intent to optimize the overall time of test runs. The tool is intended to be utilized during system or acceptance testing, primarily by technical testers.

The main area of use is execution of test scripts for web applications. Tests should be able to be executed remotely in a controlled distributed environment. A major objective has been the design and implementation of *OptiX*: a mechanism for allocating test cases to available machines in the distributed environment such that the overall execution time of a test set is attempted minimized. Further, a user interface for uploading, managing, executing and scheduling test scripts should be included, and logs from results of previous test runs should be made available. Upon a failed test, an option to report this to JIRA [1], the issue tracking system used by Altibox, should be provided.

1.4 Outline

The remainder of this thesis is organized as follows:

Chapter 2 provides a theoretical basis for the thesis by discussing some background information relevant to this thesis.

Chapter 3 introduces some essential tools and technology used throughout this project.

Chapter 4 presents an overview of OptiRun as well as the web-based interface from the user's perspective.

Chapter 5 describes the implementation of the system. Some features are explained in detail.

Chapter 6 presents OptiX: a mechanism for optimal test allocation designed for this project, as well as ORX: an alternative mechanism whose performance was used for establishing benchmark values to be used in the evaluation process of OptiX.

Chapter 7 presents the results from an experimental evaluation of OptiX, and discusses the system as a whole as well as highlighting the return on investment from a business perspective.

Chapter 8 provides some suggestions for further work.

Chapter 9 concludes this thesis.

2
Background

This chapter introduces software testing and explains some essential concepts relevant to this thesis. Further, it presents constraint programming and optimization as well as some existing related work.

2.1 Software Testing

A fundamental understanding of software testing is a useful prerequisite in order to fully understand the context of this thesis. Consequently, this section provides a short introduction to software testing. Since this is a broad field, only a small selection of relevant concepts will thus be presented.

Software testing is the process of evaluating the quality of the application, system or component being tested, commonly referred to as the test object or the system under test. Software testing may involve any and all actions oriented toward assessing the software, with the goal of determining whether it meets the required results [2].

Software is developed by human beings who can make errors. These errors may cause defects in the source code. Executing defected code can lead to failure in the program [3]. One of the purposes of software testing is to examine the test object with the intent of revealing such defects. Other objectives include to measure and ensure quality, and to provide confidence in the product [2].

2.1.1 The V-Model

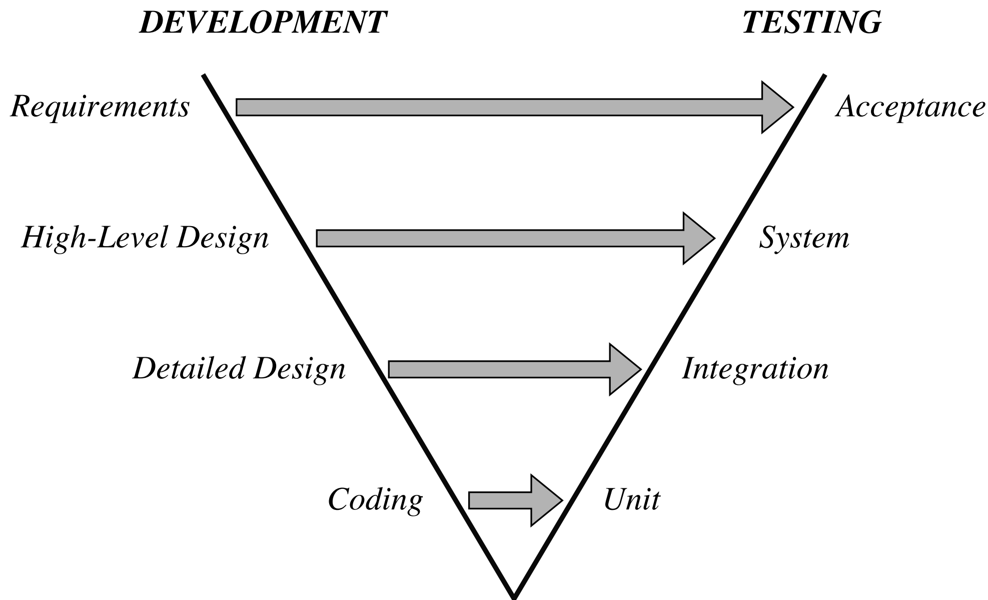


Figure 1: The V-Model

The *V-model* is an important asset in software testing, behind which the general idea is to link each development task in a software development process to an equivalent testing task of comparable importance. The development process is represented by the left branch of the letter *V*, which shows the system being gradually developed. The right branch represents the testing process in which progressively larger subsystems are tested [2].

Depending on the literary source, the V-model covers a varying number of levels. The V-model shown in Figure 1 is a general version which is created using the four levels found in [4]. The highest development level covers gathering, specification and approval of requirements. Acceptance testing correspondingly checks if these requirements are met. Further, high-level design covers the functional design of the system, and corresponds to system testing, which aims to verify if the system as a whole meets the required results. Detailed design covers technical system design and component specification, and integration testing correspondingly verifies that the different components work together as specified. The lowest level is the coding in which the specified components (modules, units and classes) are implemented. It corresponds to unit and component testing. The tests

at this level aim to verify that system components perform as specified, by testing them in isolation [2].

OptiRun is a tool for high-level test execution, and is intended for the two highest test-levels of the V-model; system testing and acceptance testing.

2.1.2 Black-Box & White-Box Testing

Software testing can be divided into *black-box* and *white-box* testing. White-box testing is based on analysis of the internal system structure, and requires knowledge of the source code. In black-box testing, the test object is seen as black-box, whose behavior is watched from the outside. The inner structure of the system is either unknown or disregarded. Test cases are determined from the specifications of the system.

Black-box testing is predominantly used for higher levels of testing. The test object is accessed through the user interface (UI). Tests are performed from the perspective of an end-user, and aim to mimic a human being interacting with the system. Black-box testing includes functional testing, which is used to validate a particular feature for correctness according to the requirements specifications [2].

OptiRun performs tests by using Selenium, an automation tool that will be introduced in Section 3.2, to automate web browsers. It executes functional UI tests on full system builds, and runs without access to or consideration of the source code or the internal system structure. The test method covered by OptiRun is therefore black-box testing.

2.1.3 Test Automation

As opposed to humans performing software tests manually, test automation is the practice of writing scripts that conduct tests when executed. Automation can be applied to tests at any level. Low-level tests, such as unit and component tests, generally run fast and are durable since they are isolated from changes in other parts of the system. As we move up to the integration level, the tests do not run as fast, and become less durable since they depend on multiple components working together in subsystems. OptiRun is concerned with testing the system from the perspective of an end-user. Such tests require the system to work as a whole, and thus generally run slower and are more brittle [5] than tests on lower levels.

There are a number of strengths to automated tests. They are superior at verifying logical functionality. They can be executed any number of times, and they run more quickly than a human interacting with the system, thus saving time and reducing effort. Time saved on manual testing can be used to increase the test coverage, which can provide reduced risk and higher software quality. Tests and tasks that would be error-prone if done manually and tests that are repeatedly performed are typical subjects for automation [6]. This includes regression testing, which is used to verify that defects have not been introduced in a new version of previously tested software [2].

Manual human testing has a number of benefits too. Human testers can identify corner cases and check how the system responds when it is used in manners it is not designed to be used for. They can also evaluate aesthetics and design of the UI as well as the overall user experience. Thus, test automation on the two highest levels in the V-model should not be a complete replacement for manual human testing; the two should instead complement each other.

Test automation is often incorporated in *continuous integration* (CI) [7] environments. This has not been done in this project, but is presented in 8.4 as a suggestion for further work.

2.2 Constraint Programming & Optimization

As explained in the previous section, the type of tests targeted in this project are time-consuming. This means that a large collections of test cases, or test sets, could take unnecessarily long time to execute if the tests were not carefully allocated among the available test machines. Minimizing the execution time of a test run has been an important objective in this project.

There are certain requirements that must be fulfilled upon allocating tests in OptiRun. A test that is specified to run in a certain browser can only be allocated to a machine with the given browser installed; this is a constraint. *Constraint programming* (CP) revolves around modelling real-world constraints by mathematical formalizations, and use them to find feasible solutions to the problem [8].

Some problems have very large sets of possible solutions, while others only have a few or none at all. Sometimes it is not enough to simply find a *possible* solution to a CP problem, as the quality of different solutions can vary greatly. In such cases, a specification of which solutions are more

valuable should be provided to help separate the good solutions from those of poor quality. The objective of the CP problem in this project is to minimize the overall execution time of a test run. An *optimization problem* is a CP problem in which such an objective is specified [8].

The optimization problem in this project does not exclusively revolve around finding an optimized allocation – in situations where the problem has a huge number of solutions, identifying the best solution can take a vast amount of time. The time taken to find the solution must therefore also be taken into account. Accordingly, some stop criteria for searching must be defined; these will be presented in Chapter 5.

Formally, the optimization problem in this project considers a set of tests $\mathcal{T} = \{t_1, \ldots, t_n\}$ with a corresponding set of durations $\mathcal{D} = \{d_1, \ldots, d_n\}$ in which the values are derived from the durations of earlier executions of each test. Further, it considers the current set of available test machines $\mathcal{M} = \{m_1, \ldots, m_m\}$. A pre-processing function p identifies the subset of machines each test can be executed on, and a function f defines the actual allocations with the overall execution time T_e. The problem seeks to minimize the total time T_t, which consists of T_e plus the searching time it takes to find the solution T_s. Thus, the expression we wish to minimize is $T_t = T_e + T_s$. Additionally, all tests in the test set must, if possible, be executed exactly once on a machine that fits any specified requirements for browser and operating system.

Some assumptions are made. Firstly, test executions are non-preemptive, so once a test execution has started, it will run uninterrupted until it completes. Further, non-cumulative execution is assumed, meaning that each test machine can only execute a single test at a time. This could have been implemented differently, but as each test requires a great deal of resources, this would prolong the execution of each test, as well as make the durations more unpredictable. Additionally, machine-independent execution time is also assumed, even though the execution times in reality may depend on aspects such as network and machine performance.

2.3 Related Work

This section presents some existing work related to this project. It opens by introducing *TC-Sched*; a mechanism optimal test case allocation and scheduling. The automated test platform *Sauce Labs* will then be introduced, before the section is wrapped up by the presentation of the *Auto-*

grader project.

2.3.1 TC-Sched

TC-Sched is a time-aware method for minimizing the execution time of test cases within a distributed system with resource constraints, proposed by Mossige *et al.* in [9]. In their paper, the authors address optimal test case scheduling. Similarly to the test allocation mechanism of OptiRun, TC-Sched aims to minimize the overall execution time of a test set by allocating tests to machines. However, it also takes into account an additional aspect of global resources. Their solution addresses situations where some tests may need exclusive access to a resource, so that two test cases which requires access to the same resource may *not* be executed simultaneously. This means that a time-aware scheduling is required to solve the optimization problem. Including this aspect in OptiRun would have been redundant, as there are no such resources involved.

2.3.2 Sauce Labs

Cloud testing services use cloud environments as a means of simulating real-life user traffic. Sauce Labs is a cloud-hosted platform for automated testing of web and mobile applications. One of the founders, Jason Huggins, were also the original creator of Selenium [10], which will be introduced in Section 3.2.

Sauce Labs bears some similarities with OptiRun. Both platforms are frameworks for automating UI tests for web applications through execution of Selenium test scripts. They both execute tests remotely in distributed environments using Selenium Grid. However, whereas Sauce Labs perform test execution on virtual machines (VMs) that are created prior to each test run and destroyed immediately after [11], OptiRun requires the use of either real, physical machines or customized VMs that remain intact after use. Because of the number of machines in OptiRun being limited to the machines connected to the distributed system, an algorithm for optimizing the overall execution time of test runs is also needed here. While Sauce Labs executes tests in their own data centers in a remote location and their own environments, OptiRun performs tests on the test object locally and in an environment intended for testing of the given test object, in this case TV Overalt. Sauce Labs is a paid service for which customers pay to gain access to using a limited number of concurrent virtual machines for a

limited amount of time each month. The optimization mechanism in this project further distinguishes the two projects.

2.3.3 The Autograder Project

In the course of the last few years, several students at the University of Stavanger have been involved in the development of the Autograder project [12] during their thesis work. It is an ongoing project that is continuously extended. Similar to OptiRun, the Autograder consists of a server backend and a web-based user interface. Its main intent is to improve student learning by executing automated tests. The Autograder is used in programming courses at the University of Stavanger, where it tests code written by students and submitted to specific repositories, aiming to evaluate whether their solutions meet the results required to pass a given programming assignment.

In contrast to OptiRun, the Autograder incorporates a continuous integration environment. As mentioned in Section 2.1.3, this has been suggested as further work in Section 8.4. Another distinction between OptiRun and the Autograder is that the latter's main use is to execute low-level tests that run quickly, as opposed to the heavier user-level tests intended for OptiRun.

One of the Autograder's most prominent strengths is its ability to give students rapid and frequent feedback on their code at any time of the day. Additionally, the teaching staff gets a clearer picture of the progress made by the students.

3 Technology

A number of tools and technologies have been applied throughout the work on this project. This chapter briefly introduces the most essential of these tools and technologies.

3.1 The Python Programming Language

The Python programming language has been used in this project. Python has efficient high-level data structures, and is known for being simple, yet powerful. Since Python programs are executed by an interpreter, it is considered an interpreted language [13].

Building this project in Python was a natural choice for several reasons. Python is compatible with Selenium, which is introduced in Section 3.2. Other reasons include offering a large set of libraries and having high-quality documentation. Details of the implementation will be discussed in Chapter 5. Version 2.7.11 of Python was used in this project.

3.2 Selenium

Selenium is an umbrella project for automation of web browsers, consisting of multiple tools and libraries. Selenium can be used to automate different types of browser jobs such as web-based administration tasks, but is primarily used for test automation [14]. The project is released under the Apache

2.0 license, and is thus free and open-source. Selenium has language bindings for several different programming languages, including Python. The Selenium tools used in this project will be introduced subsequently.

3.2.1 Selenium WebDriver

Selenium WebDriver is a tool for browser automation, largely used for test automation of websites. It interacts directly with browsers by sending calls using browser-specific WebDriver implementations that provide native support for automation [15]. There are drivers for most conventional web browsers, including Chrome, Microsoft Edge, Firefox, Internet Explorer, Opera and Safari.

The driver implementations differ among browsers. For instance, the SafariDriver must be installed as a plugin in the browser. The ChromeDriver binary and the standalone server InternetExplorerDriver must be stored locally, and their disk locations must be specified upon instantiating the driver in test scripts. The FirefoxDriver is automatically installed as a browser plugin upon installation of the Selenium WebDriver package.

Selenium WebDriver uses the drivers to create a new instance of the requested browser. It can navigate to web pages, locate UI elements and perform actions, such as clicking buttons, checking checkboxes, or populating text fields.

Selenium itself does not provide a testing module, but is commonly used with Python's *unittest* module, which is also specified at the framework of choice in the documentation of Selenium with Python Bindings [16]. Note that despite the name of this testing module, Selenium tests implementing this module are not considered unit tests from a software testing perspective, as explained in Section 2.1.1.

Listing 1 shows an example of a Selenium test script implemented as described above. When executed, this script will create a WebDriver instance using the ChromeDriver binary. A Chrome window will open, and automatically navigate to TV Overalt. It will wait for the *Login* button to appear, or for at most 10 seconds. If the button appears within the specified time-frame, it will be located and clicked. When this is done, the test is complete, and the driver closes the browser window. If all of the steps were conducted successfully, the test passes. Otherwise, the test fails, and an exception is thrown [17].

```python
import unittest
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions
from selenium.webdriver.support.ui import WebDriverWait

class Example(unittest.TestCase):
    def setUp(self):
        cls.driver = webdriver.Chrome("path/to/chromedriver.exe")

    def test_example(self):
        self.driver.get("http://tvoveralt.altibox.no/")
        WebDriverWait(self.driver, 10).until(
            expected_conditions.presence_of_element_located(
                (webdriver.common.by.By.CLASS_NAME, 'btn-login'))
        )
        login_button = self.driver.find_element_by_class_name('btn-login')
        login_button.click()

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main()
```

Listing 1: Example Selenium Test Script

3.2.2 Selenium Grid

Selenium Grid is a tool for executing Selenium tests on remote machines in a distributed environment, and thus allowing for parallel test executions on multiple machines.

A Selenium Grid environment is made up of a *hub* (master), and one or more *nodes* (slaves), all running an instance of the *Selenium Standalone Server*. The hub and the nodes interact through a JSON wire protocol [18]. Upon an execution of a Selenium test script, the hub sends the WebDriver calls specified in the test script to a node. The node then executes the browser calls locally, using a browser driver located on the node. The location of the driver on the node is specified upon starting a node, as described in Section 5.1.

Listing 2 shows how the driver can be instantiated in a test script that should be executed in Chrome on a node in the grid, as opposed to local execution in Listing 1. The script itself is executed on the hub, which maps the desired capabilities specified in the test script to a node with matching capabilities, and sends WebDriver calls to the specific node.

```
1  from selenium.webdriver.common.desired_capabilities import
       DesiredCapabilities
2
3  driver = webdriver.Remote(
4     command_executor='http://<HubHost>:<Port>/wd/hub',
5     desired_capabilities=DesiredCapabilities.CHROME
6  )
```

Listing 2: Selenium Test Script WebDriver Instantiation for Remote Execution

Reasons for wanting to incorporate Selenium Grid in OptiRun include being able to run tests against multiple browsers, browser versions and operating systems, and to reduce the execution time of test sets.

3.3 Django

Django is a high-level web development framework built on the Python programming language. It encourages rapid development and enables efficiently maintainable web applications of high quality. The framework is a free and open-sourced project maintained by the non-profit organization the *Django Software Foundation*, who describe it as fast, secure and scalable [19]. As with Selenium, Django is also essentially a collection of Python libraries [20]. The Django libraries can be imported and used to implement web applications. Additional HTML, CSS and JavaScript code can be used along with the Python code, which has been done in this project.

Aside from allowing rapid progression of development, one of the main reasons for choosing Django rather than building the dashboard from scratch or using a different web framework, is its powerful administrator site. An administrator site was exactly what was needed to build the dashboard of OptiRun. Another contributing factor was to provide consistency and the ability to communicate seamlessly with the remaining parts of the system, since Django builds on the same programming language as the rest of the system. The dashboard will be presented in Section 4.2. Details concerning the implementation will be explained in Section 5.4. Version 1.9 of Django was used in this project.

3.4 OR-Tools

In order to measure and evaluate the performance of the test allocation mechanism OptiX, which represented a major objective in this project, an alternative allocation mechanism called *ORX* was also implemented.

Detailed explanations of OptiX and ORX will be presented in Chapter 6, and a discussion, experimental evaluation and comparison of the two implementations will be presented in Chapter 7.

ORX was implemented using Google's *Operations Research Tools* (OR-tools) [21], which is an open source library for combinatorial programming and constraint optimization. The tool set is written in C++, but there are bindings for other programming languages such as Java, C# and Python. The OR-tools library strictly conforms to the Google coding styles, and is of such high quality that it has been accepted for usage internally at Google.

Listing 3 shows how a simple optimization problem is solved using this library. In this problem, a list of integers will be assigned values ranging from 0 to 2. A constraint specifying that no two identical numbers should be placed beside each other is added as a constraint. Maximizing the sum of the integers is specified as the objective. The solver searches for better and better solutions until finally arriving at an optimal solution.

```
1  from ortools.constraint_solver import pywrapcp
2
3  solver = pywrapcp.Solver('')
4  variables = [solver.IntVar(0, 2) for _ in range(3)]
5
6  for i in range(len(variables) - 1):
7      solver.Add(variables[i] != variables[i + 1])
8
9  db = solver.Phase(variables, solver.CHOOSE_RANDOM,
       solver.ASSIGN_RANDOM_VALUE)
10 objective = solver.Maximize(solver.Sum(variables), 1)
11 solver.NewSearch(db, objective)
12
13 while solver.NextSolution():
14     result = [int(item.Value()) for item in variables]
15     print result, "Sum =", sum(result)
16
17 >>> [0, 2, 1] Sum = 3
18 >>> [2, 0, 2] Sum = 4
19 >>> [2, 1, 2] Sum = 5
```

Listing 3: OR-Tools Implementation Example

4 System Overview

This chapter aims to provide an overview of how OptiRun is assembled by describing the design and architecture of the system, followed by a presentation of the web-based user interface.

4.1 Architecture

OptiRun is operated through a web-based user interface, which works as a content management system for test scripts. It also allows for sending test execution requests for the available test scripts and for scheduling test runs ahead of time.

Figure 2 shows the structural architecture of the system. In this case, the web server, the controller, the database, the file storage and the Selenium Grid hub are all located on the same machine, hence the dashed border around these elements in the figure. Some of them could, however, be separated if desirable, with little to no effort.

The web server uploads test scripts to a file location on the server. Meta data about the test cases as well as other information such as test groups, planned test executions, results and user authentication like credentials and permissions, are also stored in the database. The web server can report defects to JIRA, the issue tracking system used by Altibox.

The controller listens to test execution requests from the dashboard, and checks the schedule for planned test executions. Upon a pending test

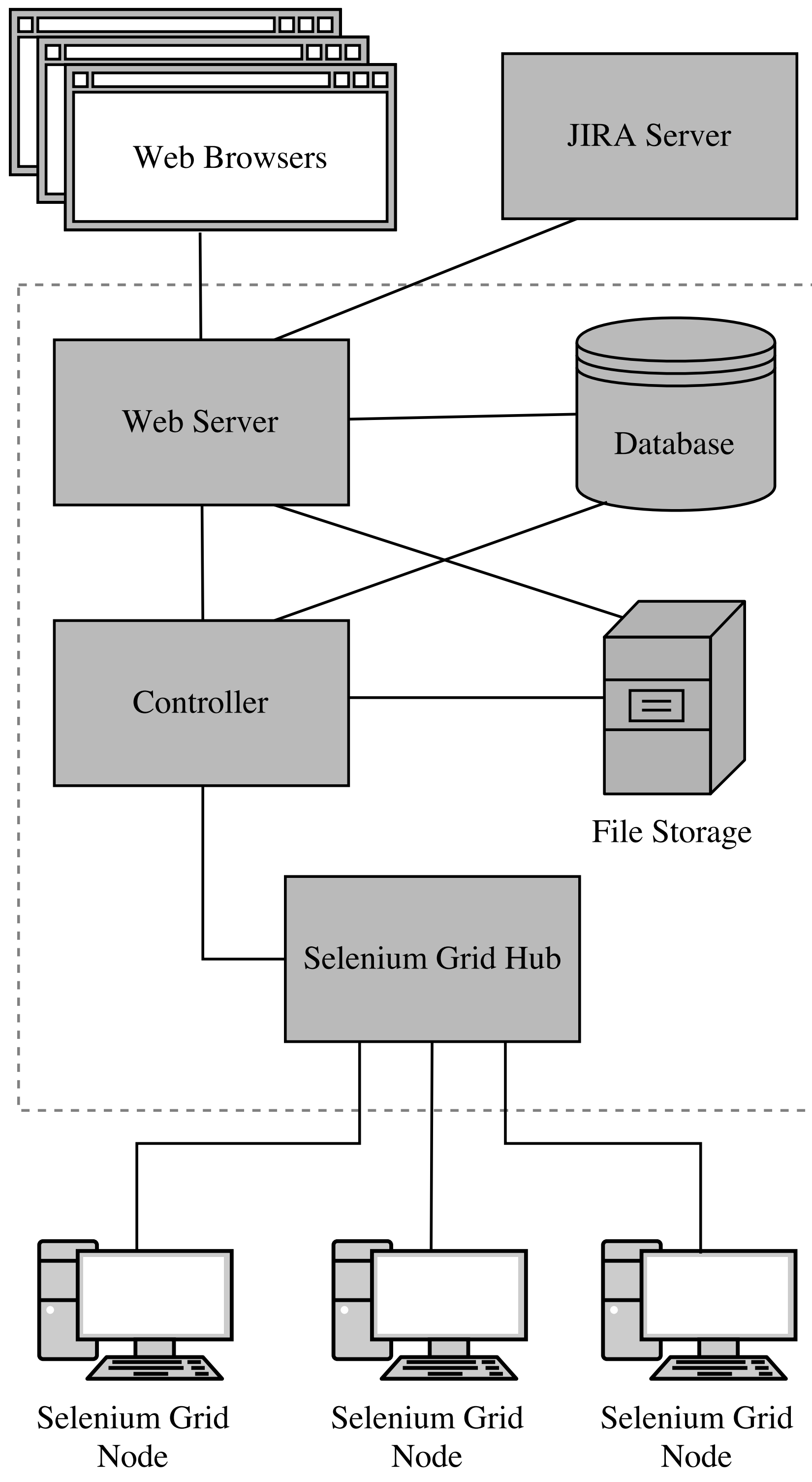


Figure 2: System Architecture

run, it fetches the relevant test scripts from the file storage. Test scripts are sent to the current instance of the Selenium Grid hub with specifications regarding which node in the grid should execute the test. After the test has finished, the controller stores the result in the database.

4.2 User Interface

The graphical user interface of OptiRun is web-based, and is built on the high-level Web framework Django. The framework provides a powerful automatic administrator interface, which has been used as a foundation for the development of the dashboard. Working with Django thus allowed for rapid development while retaining control of the content and functionality of the website.

This section presents the dashboard from the user's perspective. An in-depth presentation of how some of the functionality has been implemented can be found in Chapter 5.

4.2.1 Home Screen

After logging in, the home screen is the first thing that meets the eye of the user. This screen provides navigation to all of the modules and pages described subsequently.

4.2.2 Authentication & Authorization

The Django administrator interface provides some built-in functionality, including a module for authentication and authorization of users. *Superusers* can list, edit and create new user accounts. They can also change user information and permissions, and add users to groups.

OptiRun is a tool that is meant to be part of a business process in which only trusted peers should be allowed access. Thus, any person who wants and account must request this by someone with superuser privileges.

4.2.3 Test Case Module

All registered test cases are listed in this module. The list contains meta data about the test cases as well as some values that are retrieved or calculated using information from other tables in the database. This includes the number of times the test case has been executed, the average execution

time, the result of the previous execution and when the test case was last executed. It is possible to search and filter the list to narrow down the elements listed, and to sort the list by clicking on the attribute names in the table header. As Figure 3, shows, each list entry has a miniature bar graph aiming to provide a summary of the overall results of executions of each particular test.

New test cases can be created by clicking the button labeled *Add Test Case* in the upper right of the view. This opens a new page with a form in which the test script can be uploaded and corresponding details about the test case can be filled in. Which groups the test case should be a member of can also be specified here. Test cases can be members of multiple groups, or none. After a test case has been created, it can be edited by clicking on the title of the test case in the test case list.

An Actions dropdown menu is located above the list of test cases. Action

Figure 3: Test Case Module

options – *Delete selected test cases* and *Execute selected test cases* – can be performed on selected test cases. The former is a built-in Django function which, after the user confirms deletion in and intermediate page, removes all meta data regarding the selected items from the database. The *Execute selected test cases* action is implemented for OptiRun. The action takes the user to an intermediate page in which they can specify the platform and browsers the test cases should be executed on.

After an action has been attempted performed, a feedback message stating whether the action was successful is displayed on the screen.

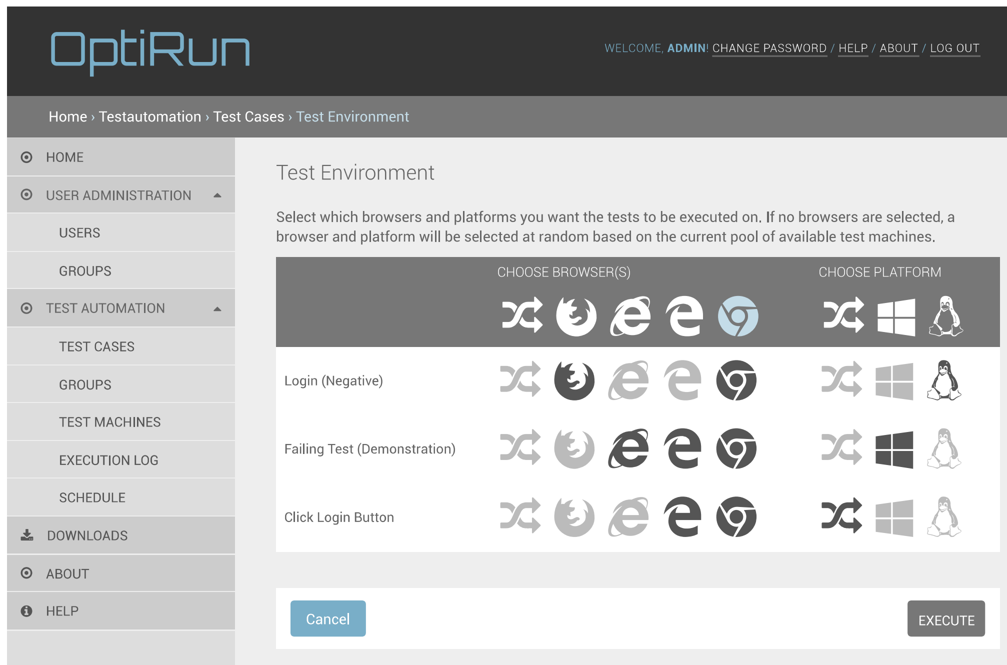


Figure 4: Intermediate Page for Immediate Test Execution Requests

4.2.4 Test Group Module

Test groups are listed similarly to test cases as described in Section 4.2.3. As with the test cases, group items can also be searched, filtered and sorted. Additionally, new groups can be created, and existing groups can be edited.

However, the group model is simpler than the test case model. The purpose of this model is to group together test cases that are often executed in the same test run, such as tests revolving around the same functionality in the test object.

4.2.5 Scheduling Module

The scheduling module provides an interface for planning test runs in the future. As with the test cases and groups, schedules are also displayed in a list can be searched, filtered and sorted. Schedule objects can be created and changed. Additionally, the schedule list enables schedule objects to be activated and deactivated from the action menu. Only activated schedule objects will start test runs according to their specified time of execution.

Upon the creation or editing of a schedule object, an execution time

can be specified along with any desired recurrence pattern along with the option of providing an end time. Test groups or individual test cases that should be included in the schedule object are also specified here.

Figure 5: Schedule List View

4.2.6 Execution Log

The layout of the execution log list is the same as the remaining list views, but the functionality differs in that log items can not be edited. Log items are created and inserted into the database based on the result from test executions. Each log object corresponds to an individual execution of a test script. As displayed in Figure 6, failed test executions can be reported to JIRA. This is done by marking the log items that should be reported and selecting *Report to JIRA* from the actions menu. When clicking on a log item to view details about the execution, there is also an attribute

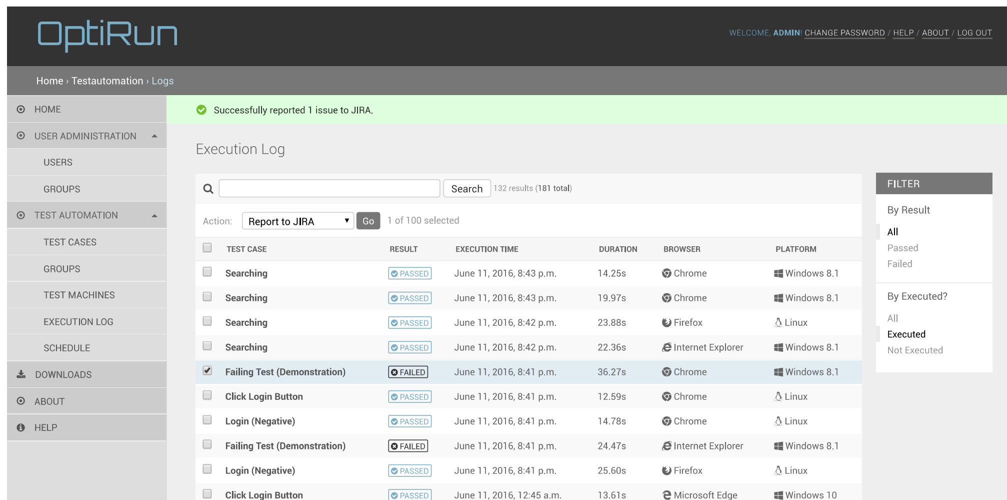


Figure 6: Execution Log List View

displaying any existing JIRA issues reported by OptiRun for the specific test, with links to the URL of the specific JIRA issues. Figure 7 shows a cropped log detail view, and displays how JIRA issues linked to a test case is listed in the log detail view. Note that this list does not correspond do an individual log item, but instead to a test case. The same list of issues will thus appear in all detail views for Log items regarding the same test case.

The information listed in the logs include a number of attributes such as the duration of the test execution, the IP address or hostname of the machine on which it was executed, as well as information such as browser, platform and test result. Additionally, the content from the standardized data streams *standard error* (stderr) and *standard output* (stdout) are also included. Stderr typically streams any automatically generated error messages or diagnostics from command line programs [22], while stdout consists of the output data from the programs themselves (if any) [23], using the *print* command in Python scripts. For better human readability, the fields are labeled *Console Log* and *Output* respectively. Figure 7 shows a detail view of a log object.

4.2.7 Test Machines

The test machines that are currently or has previously been connected to the system are listed in the test machine view, as shown in Figure 8. When

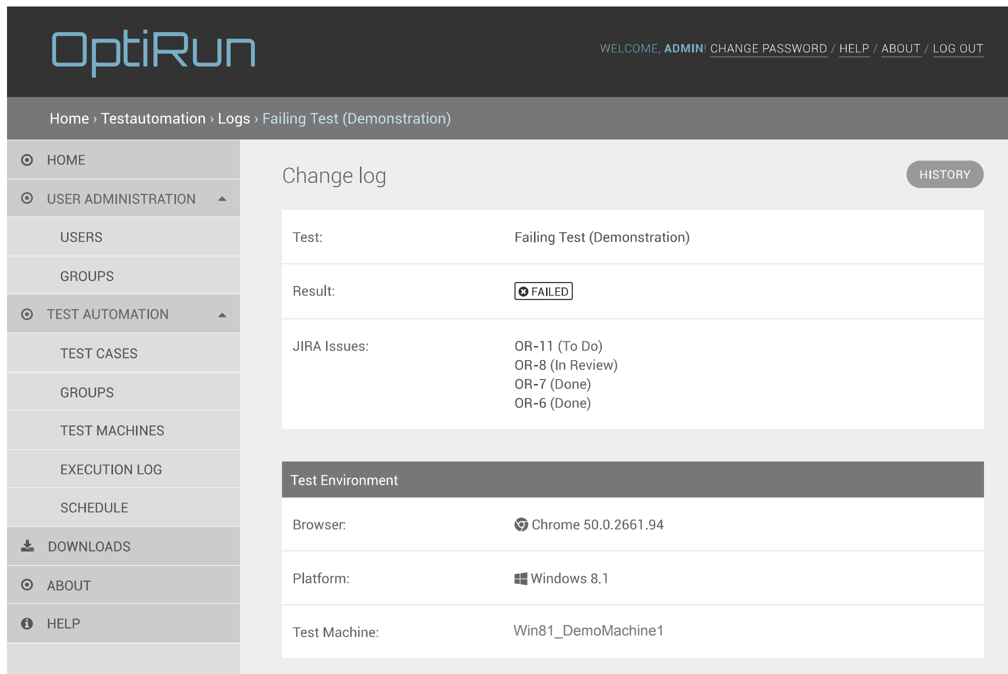


Figure 7: Execution Log Detail View

a machine is connected for the first time, details regarding the operating system and the available browsers are added to the list. The list also contains two attributes stating whether the test machine in question is currently connected to the system, and whether it has been approved for test executions. The list can be filtered based on operating system, installed browsers, activity status and approval status. Tests can not be executed on disapproved test machines. Test machine objects can be approved and disapproved either from the Actions menu or from the detail view.

4.2.8 Other

In addition to the content that has already been introduced, OptiRun also includes the following static pages:

Download Test machine packages for Linux and Windows can be downloaded from this page as well as a test script template that needs to be used with OptiRun. This page also contains setup instructions and some hints for troubleshooting if the test machine does not connect to the system as it should.

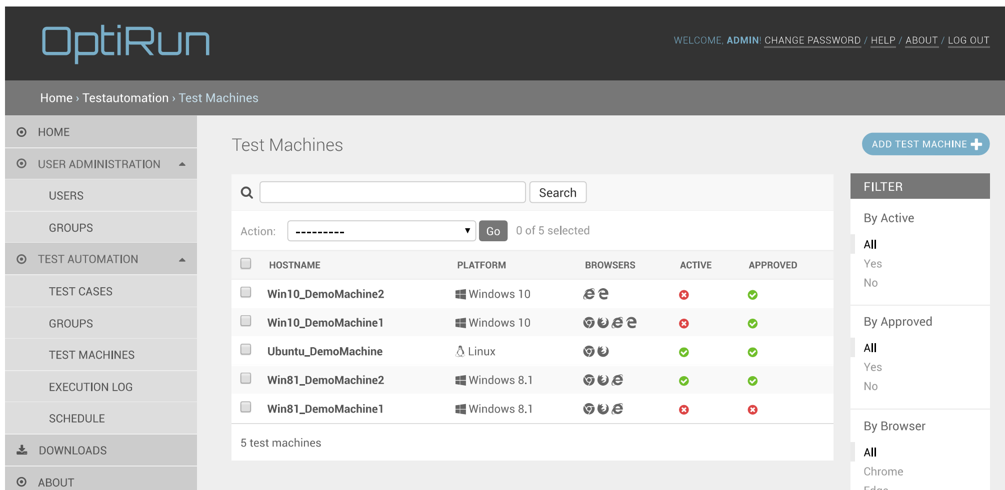


Figure 8: Test Machine List View

Help This page contains information regarding how OptiRun works, how to set it up and how to use it. This page can be of help to technical testers who wants to use the tool, and otherwise to those who wants additional insight into how it works. It includes some general information about OptiRun, some troubleshooting steps for if something is wrong with the system, and a user manual. Some of the information stated on the Download page is repeated on the Help page.

About This page provides a brief presentation of what OptiRun is and how it came to be.

5
Design & Implementation

This chapter aims to explain how OptiRun is designed and implemented. It provides a description of how the system utilizes Selenium Grid. The implementation of the controller and some important details about this part of the system are then presented. Further, the test allocation mechanism called *OptiX* is explained along with an accompanying example to illustrate how the mechanism works step-by-step. The chapter then describes how the database is structured and explains how it is accessed using Django's API for database abstraction, before finally presenting some important implementation details about the dashboard.

5.1 Selenium Grid Integration

Selenium Grid works as the backbone of all interaction between the server and the remaining machines in the distributed system of OptiRun. It is used to establish connections and to send browser calls to test machines upon test executions, as explained in Section 3.2.2.

A Selenium Grid hub can be started simply by executing the command in Listing 4, although it is possible to assign additional configuration, such as port and IP address, using flags. The default port number is 4444 for hubs and 5555 for nodes, so if nothing else is specified, these ports will be used. When the hub has started, the configuration details for the hub itself

and any connected nodes can be viewed by opening `http://<HubHost>:4444/grid/console` in a browser.

```
$ java -jar path/to/selenium-server-standalone-2.53.0.jar -role hub
```

Listing 4: Sample Shell Command for Starting Selenium Grid Hub

Starting Selenium Grid nodes, or test machines, require much longer and heavier commands, and thus more work. Also, the command must be customized, as it represents the configuration of the node and the distributed system. Therefore, a script devoted to gathering all necessary information and executing the command to start Selenium Standalone Server is included in this project. This script identifies which browsers are installed and what versions, as well as creating a unique identifier for each machine

Selenium Grid currently does not offer a documented method of specifying which machine a test should be executed on. Instead, it maps the test to a node whose configuration matches the desired specifications stated in the test, in regard to operating system, browser and sometimes even browser version. It was therefore necessary to find a way to work around this problem. This is done by utilizing a browser parameter called *applicationName*, in which additional information can be added. A 128-bit unique identifier based on the host ID, sequence number, and the current time, is created using Python's *UUID* (Universally Unique Identifier) library. By adding as a requirement to a test that the applicationName should be equal to the UUID of a specific node, the test can only be executed on the machine with that UUID.

```
$ java -jar path/to/selenium-server-standalone-2.53.0.jar -role node -hubHost
    <HubHost> -uuid 1b234276-fc02-11e5-b752-080027f8a664 -browser
    "browserName=chrome, version=49.0.2623.110,
    applicationName=1b234276-fc02-11e5-b752-080027f8a664"
    -Dwebdriver.chrome.driver=path/to/chromedriver.exe -browser
    "browserName=firefox, version=45.0,
    applicationName=1b234276-fc02-11e5-b752-080027f8a664"
```

Listing 5: Sample Shell Command for Starting Selenium Grid Node

Listing 5 shows an example of a command that will start a Selenium Grid Node with Chrome and Firefox installed, and with UUID *1b234276-fc02-11e5-b752-080027f8a664*. Once the node is connected to the hub, the configuration can be retrieved as a JSON object from `http://<HubHost>:4444/grid/api/proxy?id=http://<NodeIP>:5555`.

Along with all necessary browser drivers, the node script can be downloaded from the user interface of OptiRun. Since grid nodes need the IP address of the hub, which is dependent on the computer on which the hub is running as well as the network it is connected to, this information is added each time the controller starts. Configuration files containing the IP address of the hub, located in two separate directories – one for Windows and one for Linux – are updated when the controller starts. The directories are then automatically zipped and moved to the correct location, so that Test Machine packages downloaded from the user interface are always updated.

If it does not already exist, the Selenium Standalone Server is downloaded from a URL specified in the node script upon execution. This also happens when the controller starts.

5.2 Controller

The *controller* consists of multiple processes running in different threads. Figure 9 shows the inner structure. Each of the modules in the figure will be explained in this section.

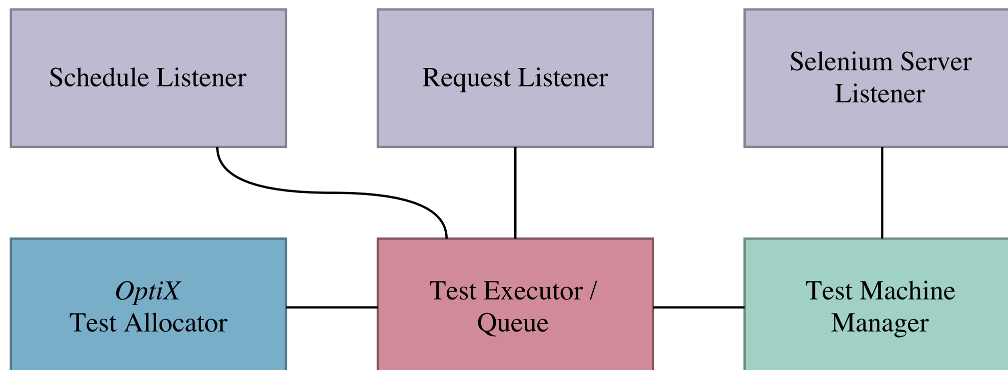


Figure 9: Controller Structure

5.2.1 Selenium Server Listener

After starting a Selenium Grid hub as described in Section 5.1 using configuration details found in the configuration file, the main job of the Selenium Server Listener is to listen to the output from the Selenium Standalone Server. The server runs as a command line program and outputs short

information and warning messages. Listing 6 shows some sample output of the server when running as a hub. The listing covers start-up output as well as node registering, test execution and node unregistering. Each output message is to the Test Machine Manager, which looks for strings similar to the bold lines in the listing, to see whether a node has been registered or marked as down.

```
# Start-up
>>> 20:44:39.073 INFO - Launching Selenium Grid hub
>>> 20:44:41.646 INFO - Will listen on 4444
>>> 20:44:41.801 INFO - Nodes should register to
    http://<HubHost>:4444/grid/register/
>>> 20:44:41.802 INFO - Selenium Grid hub is up and running
>>> 20:44:59.072 WARN - Max instance not specified. Using default = 1 instance

# Node Registering
>>> 20:44:59.094 INFO - Registered a node http://<NodeIP>:5555

# Test Execution
>>> 20:45:03.509 INFO - Got a request to create a new session: Capabilities
    [{browserName=internet explorer,
    applicationName=1b234276-fc02-11e5-b752-080027f8a664}]
>>> 20:45:03.510 INFO - Available nodes: [http://<NodeIP>:5555]
>>> 20:45:03.511 INFO - Trying to create a new session on node
    http://<NodeIP>:5555
>>> 20:45:03.511 INFO - Trying to create a new session on test slot
    {platform=WIN8_1, seleniumProtocol=WebDriver, browserName=internet
    explorer, applicationName=1b234276-fc02-11e5-b752-080027f8a664,
    version=9.11.9600.18321}

# Node Unregistering
>>> 20:45:26.240 INFO - Marking the node http://<NodeIP>:5555 as down: cannot
    reach the node for 2 tries
>>> 20:46:26.607 INFO - Unregistering the node http://<NodeIP>:5555 because
    it\textquotesingles been down for 60367 milliseconds
>>> 20:46:26.607 WARN - Cleaning up stale test sessions on the unregistered
    node http://<NodeIP>:5555
```

Listing 6: Sample Output from Selenium Standalone Server Running as a Hub

5.2.2 Test Machine Manager

As the name suggests, the responsibility of the Test Machine Manager is to manage the test machines. Upon receiving a message from the Selenium Server Listener, the Test Machine Manager checks the content of the message to identify whether a machine has connected or disconnected from the grid. Configuration for newly connected machines is retrieved as a JSON object from the URL mentioned in Section 5.1. The specified UUID as well as information regarding installed browsers and operating system are all

stored in the database.

There is a boolean *Approved* attribute in the database which is used to manage which machines are allowed to be used for test executions. This attribute is set to *False* by default for test machines that connects to the grid without being registered through the web interface. Test machine approval and disapproval can be managed from the dashboard.

The Test Executor can request a list of all live test machines, in which case the Test Machine Manager queries the database for all approved test machines, and checks whether they are connected to the grid. This is done by checking the HTML code retrieved from `http://<HubHost>:4444/grid/console`, mentioned in Section 5.1. The MIT-licensed library *Beautiful Soup* is then used to extract necessary information in order to construct a list of active, connected machines, from which non-approved machines are then removed.

5.2.3 Request Listener

All test executions managed by OptiRun are either requested for immediate execution or scheduled ahead of time. There are two different modules in the controller each handling one of these.

The *Request Listener* constantly listens to a port specified in the configuration file. When a user triggers the *Execute Now* action from the dashboard, all necessary information about the tests being requested for execution is packed as a JSON object and sent to the controller using Python's *socket* library, which communicates over TCP/IP [24].

Upon receiving a request, the request listener unpacks the JSON object and creates a list of corresponding *test* objects. The list is then forwarded to the *queue*.

5.2.4 Schedule Listener

When the Schedule Listener first starts, it retrieves schedule objects from the database. All schedule objects has a boolean *Activated* attributes which defines whether the test cases in the schedule object should be executed according to schedule. Only activated schedule objects are retrieved from the database. The Schedule Listener then calculates if and when the next occurrence of each schedule object is due. The id and next occurrence time of all activated schedule objects with a next occurrence time are added to

a list which the Schedule Listener uses to keep track of forthcoming test executions.

After the initial schedule retrieval, the Schedule Listener starts listening for schedule updates. As with immediate test execution requests, updates to schedule objects are also packed as JSON objects and sent over the network. An update is sent from the web when a schedule object is created, changed, or when the activation properties of a group of schedule objects are changed from the dashboard. The Schedule Listener receives and decodes the JSON objects, and updates the schedule list accordingly.

In an infinite while-loop in a separate thread, the Schedule Listener constantly checks whether the next occurrence of any schedule items are scheduled since the last check. If any schedule objects are due for execution, the test cases of the given schedule object are retrieved from the database, packed correctly and sent to the Test Executor. Then, the next occurrence of the schedule object is calculated, and the schedule list is updated accordingly.

5.2.5 Test Executor & Queue System

As test executions are requested by the controller, each individual test case is placed in an execution queue. The queuing system is made up of two distinct queues with descending priorities; one for immediate execution, which has the highest priority and is referred to as Q_1, and one for planned execution, which has the lowest priority and is referred to as Q_2. Which queue a test case is placed in is determined by which action triggered its execution request; if the request was received by the request listener, it is placed in Q_1, and if it was the schedule listener that identified it, it is placed in Q_2.

It is possible that a planned execution and an immediate execution of the same test case with the same browser and platform specifications is requested simultaneously. A similar situation could occur when two schedule objects that are due at the same time includes the same test case with the same specifications. Scenarios such as these could potentially lead to time and resources wasted on performing the same job more than once. To avoid this, a duplicate check is performed each time a test case is added to one of the queues. If there are duplicates, the test case is removed from the queue with the lowest priority; Q_2.

The test executor always checks Q_1 first. If there are any test cases here, the test executor empties it and handles the tests. Otherwise, it checks Q_2,

and does the same if there are any test cases waiting for execution in this queue. In other words, the test cases are not handled one by one, but in batches where the whole content of a queue is treated at once.

Starvation is a condition in which some thread fails to make progress for an indefinite period of time [25]. As previously explained, it has been decided that immediate execution requests should have the highest priority. If some test cases were requested for immediate execution again and again while there were scheduled test cases in Q_2 waiting for execution, Q_2 would be blocked from making progress, as the prioritized queue, Q_1, would be populated and then emptied repeatedly. Q_2 would then experience starvation, which was an issue that needed to be addressed.

To work around the starvation problem, it was determined that on the occasion that Q_1 was emptied, any test cases currently located in Q_2 would be moved to Q_1. Thus in addition to avoiding starvation of Q_2, the requirement of immediate execution requests being prioritized is fulfilled, as they will always be placed in the queue that will be handled first. Even so, this solution may still involve that tests requested for immediate execution can be delayed if there are a number of scheduled tests that have been moved to the Q_1.

Whenever a process is adding tests to or emptying a queue, it is important that no other process can access the queue at the same time, otherwise some tests might not be executed. This is avoided by incorporating *locks*, which are synchronization variables that can only held by one process at a time. Locks can be *acquired* and *released*, and provide mutual exclusion by ensuring that the lock can not be acquired by one process while being held by another [25]. The Test Executor requires that the lock must be acquired before proceeding to add tests to or empty a queue, and released afterward. A different lock is used to ensure that only one batch of tests is executed at a time. In addition to the Test Executor, locks have also been used in the Schedule Listener upon schedule updates, and in ORX, whose use of locks will be explained further in Section 6.2.

After test cases have been retrieved from the queue, they are allocated amongst the pool of available test machines using the OptiX allocation mechanism. This process will not be explained here, as Chapter 6 is devoted to provide a detailed presentation of OptiX as well as ORX. The execution begins once the allocation process has finished. A thread is started for each of the test machines. In these threads, each test case is started as a Python *subprocess* using a shell command in which information regarding

the desired test node and browser of the test execution are passed to the test script as arguments.

If none of the test machines match the required browser/platform specification of a given test case, it will not be executed; the test will appear in the execution log, but will be marked as *Not Executed*.

5.3 Database

The *Relational Database Management System* (RDBMS) SQLite can be regarded as a light-weight substitute to other *Structured Query Language* (SQL) based database engines. Benefits to SQLite compared to heavier database systems include it being self-contained and serverless, and the database being contained in a single disk file [26]. For convenience concerning submission of this project, SQLite was the preferred database choice. However, it can be replaced by a different SQL database engine if desirable.

Django data models define the database layout, and each model typically maps to an individual table in the database. SQL statements for creating the database itself and the tables within it are all auto-generated by Django, based on the implementations of the models. If a many-to-many relation between two different models are defined in the model implementations, a separate relationship table is created in the database to cover this.

The database can be accessed either through raw SQL queries, or through Django's own API for database abstraction. The former approach was first implemented in this system, but was then changed to the latter, as it proved to be cleaner and more consistent with the implementation of the rest of the project. It was also interesting to use a different practice of database communications than the more commonly used raw SQL queries. Listing 7 shows the equivalent of an *insert* statement as implemented conducted in OptiRun, although the listing representation is simplified, as less attributes are specified here than in reality. The implementation of the Log model is imported and then an object of this type is created with pseudo values for a small set of attributes. Line 5 in the listing represents the transaction execution and commit, where an entry in the database is created.

```
1 from testautomation.models import Log
2 from datetime import datetime
3
4 l = Log(title='test1', result=True, execution_time=datetime.utcnow())
5 l.save()
```

Listing 7: Database Communication Using Abstraction API

In addition to providing excellent readability, this abstraction greatly decreases the required number of code lines needed to achieve the same result as raw SQL statements. This is because the database location does not need to be stated, connection with the database does not need to be explicitly established before a transaction and closed when the transaction is finished, etc. The abstraction takes care of all of this. Equivalents to other query types such as *select*, *update* and *delete* are also supported through this API.

All timestamps stored in the database are in the *Coordinated Universal Time* (UTC) standard, which, as the name suggests, is universal, and therefore independent of time zones. The time zone used in the web service is set to *Europe/Oslo* in the Django settings file. Whenever a timestamp is shown on screen, it is first converted to the specified time zone using Django's *timezone* library. If the user is located in a different time zone than the one specified in the settings, a label explaining that the computer time is a given number of hours ahead or behind of server time, is displayed next to any *datetime* picker.

Storing timestamps according to the UTC standard rather than the current time zone can be considered good practice for multiple reasons. Firstly, there can be no ambiguity. Confusion and misunderstandings related to conversion across different time zones will be avoided, which also means that timestamp calculations are simple. Further, there can be no invalid dates linked to daylight savings time. Moreover, if the server were to be moved to a different time zone, timestamps would have to be converted.

5.4 Web-Based User Interface

As previously mentioned, the web-based user interface of OptiRun is built on the Django framework. Django provides a lot of built-in functionality. Most of these features can be customized either by adding new features or by overriding or extending some of the existing functionality. For instance, the administrator interface of Django projects, which is automatically generated

and will be introduced in Section 5.4.2, utilizes a standard set of CSS styling. Instead of writing new styling from scratch, it can be customized as desired. The OptiRun interface is a composition of automatic and customized content. This section aims to present the code and functionality implemented specifically for OptiRun.

Django applications are central elements in Django projects. An app in Django context refers to a set of code and functionality that is in some way related, and can include features such as static files and models [27], which will be further explained subsequently. Some apps, such as the user module, are built-in functionality in Django projects. An app called *testautomation* was created specifically for OptiRun.

A directory structure conforming to Django's guidelines is automatically generated upon creating a Django project, or a new app in an existing project. The directory structure as well as some essential files of the OptiRun project, with accompanying short descriptions, are displayed in Listing 8.

```
OptiRun/            <- Project directory
    manage.py       <- Command-line utility for administrative tasks
    db.sqlite3      <- Database
    controller/     <- All program files related to the Controller
    dashboard/      <- Python package for this Django project
        settings.py <- Project settings
        urls.py     <- Global URL dispatcher
    static/         <- Global static files (js, css, images)
    files/          <- Downloadable files (test template, test machine pckgs)
        scripts/    <- Uploaded test scripts
    templates/      <- HTML-based templates
    testautomation/ <- App directory for the Django app 'testautomation'
        models.py   <- App-specific model implementations
        admin.py    <- App-specific model representations in admin interface
        urls.py     <- App-specific URL dispatcher
        migrations/ <- App-specific database migrations
        static/     <- App-specific static files (js)
```

Listing 8: High-Level Directory Structure & Key Files of the OptiRun Project

5.4.1 Models

Data models in Django projects provide the foundation on which database tables are created and maintained. A model implementation can be seen as an equivalent to an SQL *create table* statement, but written in Python code instead. The model implementations are automatically migrated to the database by Django. Each model typically represents a table in the

database, and each model field represents a database field. Similar to SQL, the data type of each field along with any other specifications such as the maximum length of a text field, default values and help text can be passed as *field option* parameters.

```
1  from django.db import models
2
3  class TestCase(models.Model):
4      title           = models.CharField(max_length=80)
5      script          = models.FileField(upload_to='scripts')
6      description     = models.TextField()
7      groups          = models.ManyToManyField('Group')
8      schedules       = models.ManyToManyField('Schedule')
```

Listing 9: Model Implementation

Listing 9 shows a reduced adaption of how the test case model has been implemented. This simplified model representation contains four model fields. The *script* field is of the type *FileField*, and the directory that the files should be uploaded to is passed as a parameter. The file upload itself is taken care of by Django.

5.4.2 Admin

An important asset of Django is the automatic administrator interface. The administrator interface reads meta data of models from the database, based on the implementations in *admin.py*, where the functionality as well as specifications regarding what content should be available, and how, is specified.

```
1  from django.contrib import admin
2  from .models import TestCase
3
4  class TestCaseAdmin(admin.ModelAdmin):
5      list_display = ('title', 'execution_count')
6      search_fields = ['title',]
7      fields = ('title', 'script', 'description', 'groups', 'schedules')
8
9      def execution_count(self, obj):
10         return Log.objects.filter(
11             test_id=obj.id).exclude(result__isnull=True).count()
12
13 admin.site.register(TestCase, TestCaseAdmin)
```

Listing 10: Implementation of Model in Administrator Interface

Listing 10 shows a simplified interpretation of how the test case model

is represented in the administrator interface. This adaption builds on the model implementation from Listing 9. Model administrator representations are subclasses of *ModelAdmin*, and specifies how the model should be represented in the administrator interface. This interpretation specifies values of three of the many ModelAdmin options; which fields of the model should be displayed in the overview list of the test case module (*list_display*), which fields should be searchable (*search_fields*) and which which fields should be present in the creation/change form (*fields*). Line 9 registers the model to the administrator interface with the specifications stated in the TestCaseAdmin class.

In the actual implementations of the model representations in the administrator interface, a number of additional fields and specifications are also included. Custom form validation functionality can be integrated with the creation/change form. This has been done with test case objects to ensure that only Python scripts can be uploaded. Admin actions that can be performed on multiple model objects simultaneously are also implemented here. The most significant admin actions in OptiRun is the *Execute Now* action for test cases and the *Report to JIRA* action for log objects.

5.4.3 Issue Tracker Reporting

As earlier mentioned, Atlassin's JIRA is the issue tracker software used by Altibox. The issue tracker reporting supported by OptiRun is therefore built on JIRA's *REST* (Representational State Transfer) API.

The JIRA integration is used for two things: reporting failed test executions and retrieving issues linked to specific tests in the execution log. Failed test executions can be reported to JIRA by marking them in the log list and selecting *Report to JIRA* from the actions menu. Any duplicates or log entries that did *not* fail are removed from the list. If the JIRA server is unavailable, an error message will be displayed. Otherwise, the REST API is used to search for JIRA issues linked to each log entry in the list. If there are any existing open issues for a log entry, a comment is added to the comment section of the issue, saying that the problem has been reproduced, and including details about the failed execution. Otherwise, a new issue is created. The issue descriptions are formatted programmatically with the intent to provide helpful, readable information. Figure 10 shows an example of how a comment or issue description created by OptiRun looks in JIRA.

OptiBot added a comment - 36 minutes ago

Reproduced 11.06.2016 18:41.

Test	Failing Test (Demonstration)
Machine	Win81_DemoMachine2
Browser	Chrome 50.0.2661.94
Operating System	Windows 8.1

Console Log:

```
E
======================================================================
ERROR: test_nonexistent_class (__main__.NonexistentClassTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\OptiRun\dashboard\files\scripts\nonexistent_class_test.py", line 28, in test_nonexistent_class
    wdw(driver, 10).until(EC.presence_of_element_located((webdriver.common.by.By.CLASS_NAME, 'non-existent')))
  File "C:\Python27\lib\site-packages\selenium\webdriver\support\wait.py", line 80, in until
    raise TimeoutException(message, screen, stacktrace)
TimeoutException: Message:
Screenshot: available via screen
Stacktrace:
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0 (None:-2)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance (None:-1)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance (None:-1)
    at java.lang.reflect.Constructor.newInstance (None:-1)
    at org.openqa.selenium.remote.ErrorHandler.createThrowable (ErrorHandler.java:206)
    at org.openqa.selenium.remote.ErrorHandler.throwIfResponseFailed (ErrorHandler.java:158)
    at org.openqa.selenium.remote.RemoteWebDriver.execute (RemoteWebDriver.java:678)
    at org.openqa.selenium.remote.RemoteWebDriver.findElement (RemoteWebDriver.java:363)
    at org.openqa.selenium.remote.RemoteWebDriver.findElementByClassName (RemoteWebDriver.java:477)
    at org.openqa.selenium.By$ByClassName.findElement (By.java:391)
```

/OptiRun

Figure 10: JIRA Issue Description

```
from jira import JIRA

jira_instance = JIRA(
    options={
        'server': jira_server,
        'verify': False,
        'get_server_info': False
    },
    basic_auth=('<Username>', '<Password>')
)

# JIRA Issue Search
issues = jira_instance.search_issues('<Search String>')

# JIRA Issue Commenting
jira_instance.add_comment(issues[0].id, '<Comment>')

# JIRA Issue Creation
jira_instance.create_issue(
    project='NGTV',
    summary='OptiRun: <Test Case Title> (<Test Case ID>) FAILED',
    description='<Issue Description>',
    issuetype={'name': 'Bug'},
    components=[{'name': 'web'}]
)
```

Listing 11: Issue search, commenting and creation using the JIRA REST API

In the log detail view, which is accessed by clicking on a log entry, there is a field displaying a list of clickable JIRA issues linked to the given test case, if any, and their respective statuses. As with issue reporting, an error message is shown if contact with the JIRA server could not be established.

Listing 11 shows how issue search, commenting and creation are performed using the JIRA REST API. The integration with the JIRA REST API is located in *admin.py*, and is called from the methods *report_to_jira* and *get_jira_issues* in the *LogAdmin* class of this file.

5.4.4 Event Recurrence

Rrule (Recurrence Rule) is a module in the Python library *dateutil*, which provides an extension to Python's *datetime* module [28]. It is a small and fast library used in OptiRun to allow recurrence patterns of schedule objects. Rrule instances can be implemented multiple ways. In this project, it is achieved through passing a string with a specific format, containing information about the desired recurrence constraints. This string is stored in the database, and can be used at any point to create an rrule instance in order to inquire when the next event should take place. An example of such a string, how rrule instances are created in this project, and how the next occurrence is retrieved, can be seen in Listing 12.

The string in the listing above is used to create an rrule instance in which the first occurrence is set to the 15th of June 2016 at 3 PM, and repeats weekly. The output shown in the listing is valid if the script was executed before this date, otherwise it will produce a different output In addition to the recurrence properties shown in the listing above, the library provides an extensive number of recurrence options, including end date, occurrence count and interval.

```
1 from dateutil.rrule import rrulestr
2 from datetime import datetime
3
4 rule_string = "DTSTART:20160615T150000\nRRULE:FREQ=WEEKLY"
5 rule = rrulestr(rule_string)
6
7 print rule.after(datetime.now())
8
9 >>> 2016-06-15 15:00:00
```

Listing 12: Recursion Rule

The rrule strings in this project are built dynamically when a schedule

object is created or edited, and then stored in the database as attributes to entries in the schedule table. In the test case, group and schedule modules of OptiRun, recurrence rule strings are used to create rrule instances, which again are used to find out the time of the next planned execution for the particular test case, group or schedule. Rrule instances are also created by the controller to check when the next test run is scheduled.

6
Test Allocation Mechanism

As explained in Chapter 2, the type of tests intended for OptiRun generally runs slowly. Therefore, the design and implementation of a mechanism that would efficiently allocate tests to machines in an attempt to minimize the duration taken to search for an optimal allocation and execute the test set has been a major objective in this project. This mechanism has been named OptiX, and will be thoroughly explained in Section 6.1. As explained in Chapter 2, an alternative allocation mechanism has also been implemented using Google's OR-tools library. The alternative mechanism has been named ORX, and will be presented in Section 6.2. ORX was used to establish benchmark values for the evaluation process of OptiX. The results will be presented and discussed in Section 7.1.

6.1 OptiX

The OptiX allocation mechanism consists of a sequence three steps; sorting, initial allocation and enhancement iterations. It can be seen as an extended greedy algorithm, and will be explained in this section.

In order to give a better understanding of how the algorithm works, a demonstration example will be used throughout the explanation. In this example, we assume that there are three test machines connected to the system, as well as the test set listed in Table 1. Note that the durations of the tests intended for OptiRun generally range between approximately

30 and 120 seconds, but this example deliberately uses shorter durations to better illustrate how the mechanism works.

Test	**Duration**	**Executable on**
t_1	6s	m_1, m_2, m_3
t_2	3s	m_1, m_2, m_3
t_3	8s	m_1, m_2, m_3
t_4	5s	m_1, m_2, m_3
t_5	3s	m_1, m_2, m_3
t_6	4s	m_1, m_2, m_3
t_7	7s	m_1
t_8	3s	m_2
t_9	9s	m_3
t_{10}	5s	m_1, m_3

Table 1: Example Test Set

6.1.1 Preliminary Sorting

The first step is a preparation for the initial allocation. In a scenario where toward the end of the allocation, all tests waiting to be allocated could be executed only on one specific machine, the overall execution time could be greatly affected. Similarly, if toward the end of the initial allocation the last test to be allocated is estimated to have a This step seeks to avoid such scenarios by strategically ordering the tests before the initial allocation starts. The sorting does not play a crucial role in the allocation mechanism, but it creates an excellent starting point as the tests are sorted in a way that makes them well suited for the initial allocation in the next step.

There are two criteria to the sorting process: the first criteria is the number of machines each test can be executed on, in ascending order, and the second criteria is estimated test duration in descending order. In Python this can be done in a single line of code, as shown in Listing 13.

```
1 tests.sort(key=lambda test: (len(test.executable_on), -test.duration))
```

Listing 13: Python Code for OptiX Sorting Step

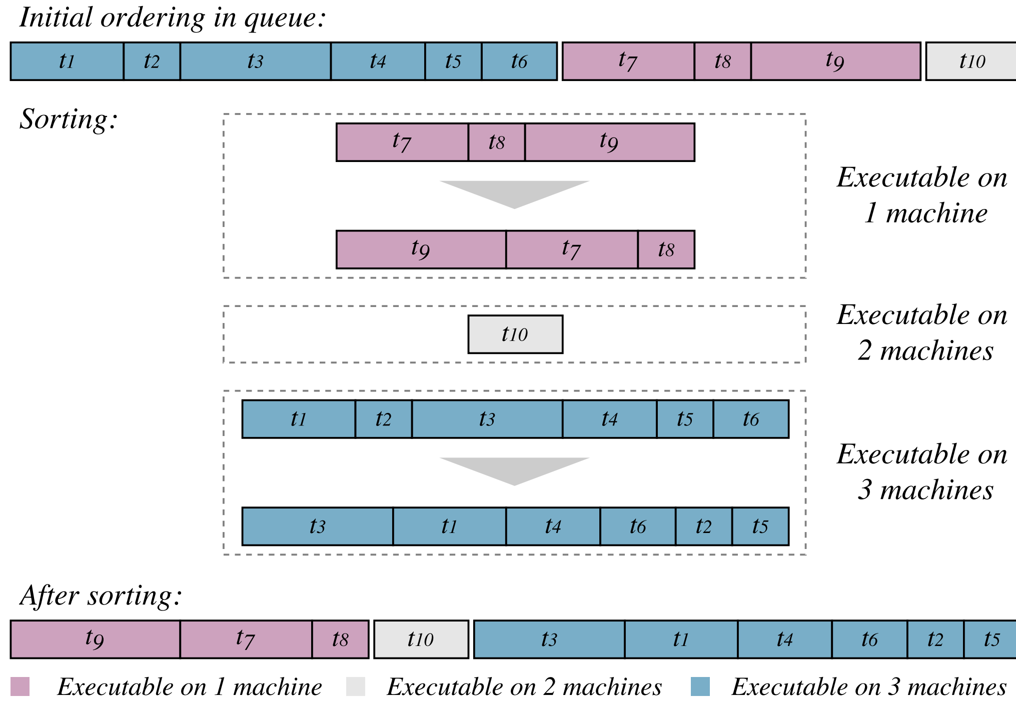


Figure 11: Sorting of Example Test Set

Figure 11 illustrates how the sorting works with the example test set introduced earlier. Since t_7, t_8 and t_9 can only be executed on a single machine each, these are placed first in descending order of their duration. Test t_{10}, which can be executed on two machines, comes thereafter. Finally, tests t_1 through t_6, which can run on all of the machines, are ordered and added to the list of tests.

6.1.2 Initial Allocation

The initial allocation is a greedy algorithm, which means that it always makes a locally optimal choice in the hopes that it will give the best result in the end [29]. Greedy algorithms are powerful and work well for an extensive spectrum of problems. Creating a greedy algorithm was a suitable choice for this problem.

This step creates the foundation of the test allocations. After being sorted, the list of tests is iterated through, and the tests are allocated one

Algorithm 1 *OptiX*: Initial Allocation Step

Require: $length(test.executableOn) \neq 0$ **for** $test \in tests$

1: **function** InitAllocation($tests$, $numMachines$)
2: $allocations \leftarrow \varnothing$
3: $durations \leftarrow \big[0$ **for** $i \in 0 : numMachines - 1\big]$
4: **for** $test \in tests$ **do**
5: $selectedMachine \leftarrow test.executableOn\big[0\big]$
6: **for** $x \in test.executableOn$ **do**
7: **if** $durations\big[x\big] < durations\big[selectedMachine\big]$ **then**
8: $selectedMachine \leftarrow x$
9: $durations\big[selectedMachine\big] \leftarrow test.durations$
10: $allocations\big[test.id\big] \leftarrow selectedMachine$
11: **return** $allocations$

by one, starting with the *longest* of the tests that are executable on the *fewest* machines. The initial allocation thus uses a slightly altered *Longest Job First* (LJF) approach, which provides quite a lot more flexibility toward the end of the allocation compared to what would be accomplished using a *Shortest Job First* (SJF) approach. Each test is initially allocated to the machine that currently has the shortest overall duration among the machines that the test is executable on.

Algorithm 1 displays the initial allocation of OptiX represented with pseudo code. The function requires that any tests whose desired test environment does not match with any available test machines have been removed from the test list. It also assumes that the test elements in the list have three attributes: an id, a duration and a list containing the indices of the machines it can be executed on. These measures are taken care of in the preprocessing that occurrs before the OptiX mechanism begins.

Figure 12 shows how the test set is initially allocated among the test machines. Tests t_9, t_7 and t_8 are first allocated to m_3, m_1 and m_2 one by one, as each of them can only be executed on that machine. Test t_{10} can be executed on both m_3 and m_1, but as the latter currently has the shortest overall execution time, this is the machine it is allocated to. The remaining tests are allocated in the same way. After the initial allocation is finished, the overall execution time is 19. However, the total durations among the machines are slightly uneven, so there might be room for improvement. This will be examined in the last step of OptiX.

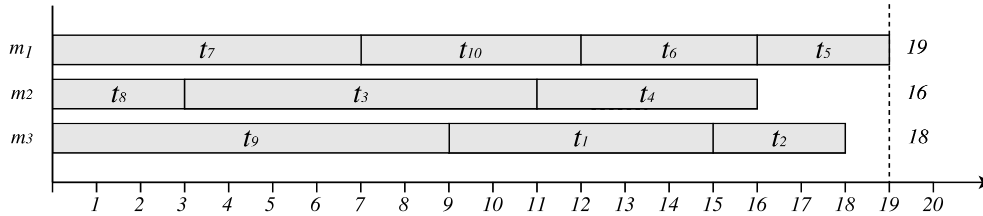


Figure 12: Initial Allocation of Example Test Set

6.1.3 Improvement Iterations

Greedy algorithms are simple, yet efficient, and provide adequate results most of the time. However, they do not always yield optimal solutions. In order to improve the result, an additional step has been included in the OptiX. This is the most complex element in the process.

The basic idea behind the iteration step is to find subsets of tests among the machine with the longest execution time and the other machines, and swap the two subsets that will result in the largest reduction in duration. The subject of each iteration is the machine that currently has the longest execution time. In the case of the example test set, this is m_1. This will be referred to as the *swapper*. Each of the remaining machines are addressed one by one, starting with m_2, which is a *swappee* candidate. The tests that are currently allocated to the swapper, but also can be executed on the swappee candidate are, retrieved. In the example, these are t_5 and t_6. A list of all possible subsets from this set is then created using Python's *itertools* library. The same is done the other way around, with t_3 and t_4 for the swappee candidate. After this, OptiX examines how each swap would affect the overall duration of the two machines in question. After comparing all subsets, it is concluded that the best swap is t_5 and t_6 from m_1 for t_4 from m_2, which will decrease the overall duration among the two machines with 1 second. Since m_3 can not provide any better options, this swap is conducted. This process will repeat until there is no way to improve the allocations, which in this case is after one swap.

Figure 13 shows how the subset swap is conducted, and how it affects the overall duration. Although it is not guaranteed to happen, OptiX found an optimal solution to the optimization problem in this example. ORX also found a solution that provided 18 seconds, even though the exact allocations differed slightly.

Upon evaluating each swappee candidate, a naive best-case duration is

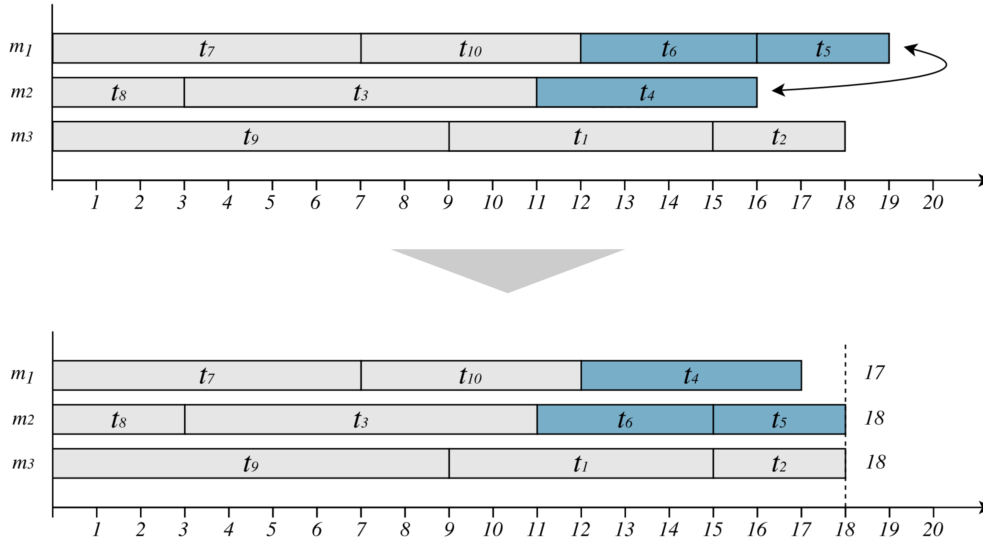


Figure 13: Enhancement Iteration of Example Test Set

calculated by adding the durations of all the tests from the two machines and dividing the sum by two. If the time used to search for the best swap exceeds the difference between the total duration of the swapper (here: 19), and this best scenario duration (here: $\frac{19+17}{2} = 18$), which in this case is 1, we will continue to the next swappee candidate. This means that we will examine the subsets of m_2 for at most 1 second.

Sometimes, however, we want to stop the searching earlier, as continuing is no longer beneficial. For that reason, two stop criteria are introduced:

1. Prior to the first enhancement iteration, a naive best-case overall duration value is calculated by dividing the sum of all test durations by the number of machines available. Once the time difference between this value and the maximum machine duration is exceeded, the iteration process will stop.
2. There is a timeout set to 30 seconds. If the iterations still are not finished at this point, the method will be terminated, and the allocations found up to that point will be kept.

During the development phase, a clear problem stood out; creation of subset lists from extremely large interchangeable test sets between two machines could take several minutes, and sometimes even lead to memory leaks so that the whole system crashed. This was because there was simply

too many possible subsets. To work around this problem, a restriction of the maximum size of the subsets had to be established. Through trial and error, the following was determined, where x is the maximum size of each subset:

$$f(x) = \begin{cases} x & \text{if} \quad x < 10 \\ 20 - x & \text{if } 10 \leq x < 20 \\ 1 & \text{if } 20 \leq x \end{cases}$$

Thus, if the number of interchangeable tests are 19 or more, the subset list will only consist of single tests in addition to the empty set. Although not optimal, this was a compromise that helped resolve the problem effectively. This means that for large interchangeable sets, tests can only be swapped one against one or moved from one machine to another. It is therefore reasonable to think that better swaps might exist, although identifying these would take too much time and potentially lead to memory leaks.

Because of the stop criteria described earlier and the subset size restriction, OptiX cannot guarantee to find an optimal solution to the allocation problem. However, as explained in Section 2.2, the objective of the optimization problem is to minimize the total time, which means that the time used to search for the best solution is also of high importance, and should be prioritized as such.

Another consequence of the stop criteria is that the mechanism becomes non-deterministic. Two different executions of the same job may not take the exact same time, which means that if OptiX is tested against the same allocation problem twice, it might time out at different points, and thus produce different results.

Algorithm 2 lists a high-level representation of the improvement iteration step of Optix. Although somewhat simplified, it manages to capture the essence of how this step is implemented. The symbols α and β represents the swapper and the swappee candidate respectively, and are used to separate the two whilst avoid using a profuse amount of cumbersome and similar variable names. The subscripts refers to properties that are either implicit or calculated in a trivial way unrelated to the core of this step, with the exception of the GetSubsets method, which was explained earlier.

Algorithm 2 *OptiX*: Improvement Iteration Step

```
 1: function ITERATE(tests, allocations, numMachines)
 2:     while True do
 3:         α ← ⟨index of machine with longest duration⟩
 4:         α_subsets ← GETSUBSETS()
 5:         dur_longest ← α_dur
 6:         for i ∈ 0 : numMachines − 1 | α do
 7:             β ← i
 8:             β_subsets ← GETSUBSETS()
 9:             for α_subset ∈ α_subsets do
10:                 for β_subset ∈ β_subsets do
11:                     α_tmpDur ← α_dur − α_subsetDur + β_subsetDur
12:                     β_tmpDur ← β_dur − β_subsetDur + α_subsetDur
13:                     if α_tmpDur < dur_longest and β_tmpDur < dur_longest then
14:                         dur_longest ← max(α_tmpDur, β_tmpDur)
15:                         swapperSubset ← α_subset
16:                         swappeeSubset ← β_subset
17:                         swapper ← α
18:                         swappee ← β
19:                 if ⟨stop criteria reached⟩ then
20:                     break
21:             if ⟨stop criteria reached⟩ then
22:                 break
23:         if ⟨improvement found⟩ then
24:             for test ∈ swapperSubset do
25:                 allocations[test.id] ← swappee
26:             for test ∈ swappeeSubset do
27:                 allocations[test.id] ← swapper
28:         else
29:             break
30:     return allocations
```

Each iteration is not guaranteed to affect the overall duration, only reduce the duration of the two involved machines. For instance, imagine a scenario where both machine A and machine B have a total duration of 100 seconds, and machine C only has a duration of 80 seconds. Machine A is then identified by OptiX as one of the machines with the longest duration. OptiX might be able to find a swap which results in machines A and C both having a duration of 90 seconds. Even so, the overall duration of the system is not affected, as machine B still has a duration of 100 seconds. Another iteration is thus required to search for an improvement for machine B.

6.2 ORX

The implementation of ORX is of the same style as the OR-tools example shown in Listing 3 in Chapter 3, but more complex. ORX takes two lists; one containing the durations of the tests, and another containing which machines each test is executable on. OR-tools does not support decimal values, so as opposed to OptiX, ORX regards all test durations as integer values. The *executable on* list is nested, with one list belonging to each test. These inner lists contain binary values representing whether or not the test can be executed on a given machine. It is then added as a constraint that each test should be allocated to exactly one machine, and as the objective that the overall duration should be minimized.

However, there was one major issue upon the implementation of ORX. It was not possible to interrupt or time out the *NextSolution* method. For very small data sets, this was not a problem, but once the data sets grew slightly bigger and the combination of possibilities grew rapidly, the method could take hours. This meant that a set of stop criteria had to be introduced. However, the problem could still not be solved by running the solver in a separate thread, as there is no integrated method of terminating regular threads in Python to the best of the author's knowledge. The problem was solved by introducing Python's *multiprocessing* library.

The solving method is started as a *Process* from the multiprocessing library. In order to allow two processes to share lists, a *Manager* is needed, and to ensure synchronization, a *Lock* has been used. Listing 14 shows how these multiprocessing modules are used in ORX.

ORX will be terminated if one of the following stop criteria are met:

- The search for an improvement has taken 500 times as long as it took

to find the previous improvement.

- The previous improvement took 50 times as long to find than the improvement itself.
- 30 seconds have passed (timeout).

The author explored using smaller and larger values for the break conditions of the two former stop criteria, but were not able to identify values that generally provided better results.

```
1  from multiprocessing import Process, Manager, Lock
2
3  manager       = Manager()
4  allocations   = manager.dict()
5  max_durations = manager.list()
6  last_updated  = manager.list()
7  lock          = Lock()
8
9  # ...
10
11 p = Process(target=find_solution, args=(durations, executable_on,
       allocations, max_durations, last_updated, lock))
12 p.start()
13
14 while p.is_alive():
15     if lock.acquire():
16         if <stop criteria fulfilled>:
17             p.terminate()
18             break
19         lock.release()
20
21 # ...
22
23 def find_solution(self, durations, executable_on, allocations,
       max_durations, last_updated, lock):
24     # ...
25
26     while solver.NextSolution():
27         lock.acquire()
28         # Store findings in the shared lists 'allocations', 'max_durations'
       and 'last_updated'
29         lock.release()
```

Listing 14: ORX Multiprocessing

7 Evaluation

This chapter opens with introducing a collection of test sets used to measure the performance of the allocation mechanism OptiX through experiments, and compare it to the performance of ORX on the same collection. The results are evaluated and discussed, followed by an assessment of factors that may threated the validity of the results obtained in the experimental phase. OptiRun is then discussed and evaluated as a whole, before the chapter rounds off by discussing the return on investment put into this project.

7.1 Experimental Evaluation

In order to measure and evaluate the performance of OptiX, which represented a major objective in this project, an experimental evaluation was performed on OptiX as well as on ORX to establish benchmark values.

7.1.1 Test Data

The test data used in the experimental evaluation is divided into four collections, each consisting of three pseudo test sets. The test sets in each collection all represent scenarios with a given number of tests and available machines. What separates the test sets in the same collection is the number of machines each test in the test set can be executed on

Collection	**Tests**	**Machines**	**Test Set**	**No. of Machines Tests are Executable On**
c_1	1000	100	ts_1	100
			ts_2	10
			ts_3	*Random*
c_2	1000	10	ts_4	10
			ts_5	5
			ts_6	*Random*
c_3	200	50	ts_7	50
			ts_8	10
			ts_9	*Random*
c_4	200	10	ts_{10}	10
			ts_{11}	5
			ts_{12}	*Random*

Table 2: Test Data Used in Experimental Evaluation on a Logarithmic Scale

Each collection contains one test set in which every test can be executed on every machine, one where the tests can only be executed on a small selection of the machines and one where the number of machines each test can be executed on varies and is determined at random. This means that there is also a varying number of combinatorial solutions to the optimization problem.

Table 2 shows details about the test data which is randomly generated; only the number of tests, test machines and how many machines each test is executable on was specified upon generation. Each test was assigned a duration between 30 and 120 seconds, and a set of machines on which they were executable on, both generated at random. The test data is designed to imitate realistic scenarios, although somewhat amplified. Using larger test sets than ts_1 through ts_3 would not be very meaningful, as 1000 tests and 100 test machines are very large numbers in this context, and are not likely to be exceeded any time soon.

	OptiX			**ORX**		
Test Set	T_s	T_e	T_t	T_s	T_e	T_t
ts_1	1.06s	739.00s	**740.06s**	34.93s	1153.00s	1187.93s
ts_2	1.04s	748.00s	**749.39s**	34.25s	1224.00s	1258.25s
ts_3	22.32s	730.00s	**752.32s**	34.44s	1172.00s	1206.44s
ts_4	0.02s	7371.00s	**7371.02s**	30.37s	7797.00s	7827.37s
ts_5	0.04s	7496.00s	**7496.04s**	30.27s	7569.00s	7599.27s
ts_6	0.16s	7355.00s	**7355.16s**	30.26s	7773.00s	7803.26s
ts_7	0.20s	299.00s	**299.20s**	30.39s	365.00s	395.39s
ts_8	0.36s	309.00s	**309.36s**	30.27s	338.00s	368.27s
ts_9	1.17s	305.00s	**306.17s**	30.36s	319.00s	349.36s
ts_{10}	0.02s	1427.00s	**1427.02s**	30.06s	1435.00s	1465.06s
ts_{11}	0.61s	1495.00s	**1495.61s**	30.06s	1553.00s	1583.06s
ts_{12}	0.79s	1521.00s	**1521.79s**	30.06s	1531.00s	1561.06s

Table 3: Experimental Test Results

Both OptiX and ORX were tested with each of these 12 pseudo test sets. The results can be found in Table 3, which provides searching time, execution time and total time obtained with both allocation mechanisms for each test set. As explained in the formal definition of the optimization problem in Chapter 2, the objective of the problem was to minimize the $T_t = T_e + T_s$, that is to say the searching time used to find the solution plus the execution time used to execute the tests.

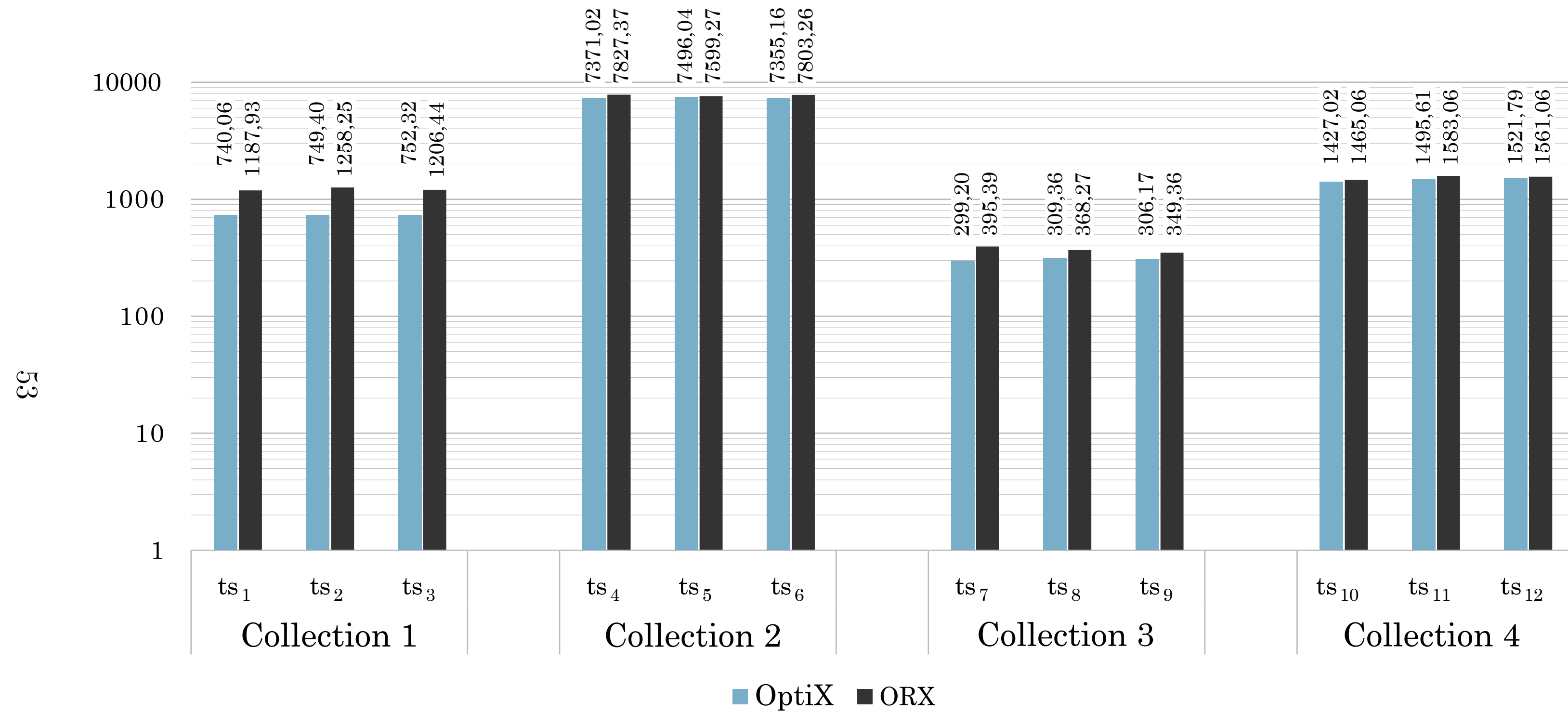


Figure 14: Complete Results from Experimental Evaluation on a Logarithmic Scale

The complete results from the experimental evaluation are visualized in Figure 14. Because of major variations in numbers, a logarithmic scale is used in the graph. The results from each collection will subsequently be discussed individually.

7.1.2 Test Data Collection 1

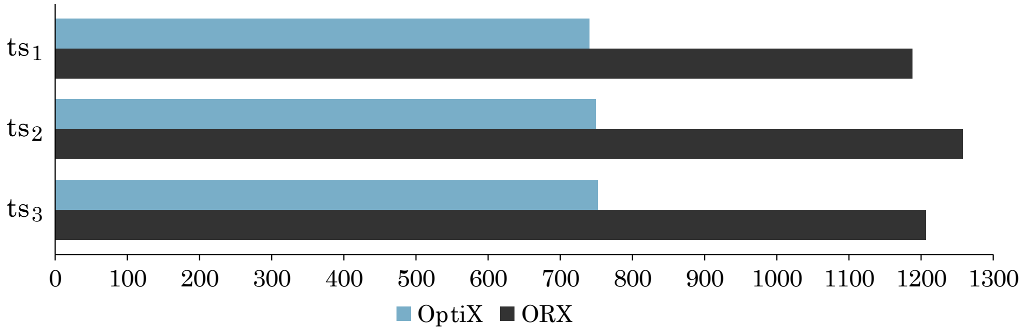


Figure 15: Results from Test Data Collection 1 in Experimental Evaluation

	Diff. from ORX	% of ORX	Iterations	Initial T_e
ts_1	447.87s	62.30%	2	739s
ts_2	508.85s	59.56%	9	776s
ts_3	454.12s	62.36%	81	790s

Table 4: Details from Collection 1 in Experimental Evaluation

The first collection of test sets consisted of ts_1, ts_2 and ts_3, with 1000 tests and 100 test machines. This collection was designed to test the mechanisms in situations with a large number of both tests and machines, and thus an abundance of possible solutions.

As Figure 15 clearly indicates, OptiX provided excellent results compared to ORX for these 3 test sets. Not only were the T_t values provided by OptiX between 447.87 and 508.85 seconds faster than ORX; it also provided better searching times. Although the searching process of ORX timed out after 30 seconds, the whole process took just over 34 seconds in all of these cases, which can likely be explained by the time required to extract the results being extended as a consequence of profuse amounts of tests and machines. This was the collection in which the results provided by OptiX was the most prominent compared to those of ORX. All 3 test sets in this

collection have a very large number of possible solutions, which is likely a contributing factor as to why the effectiveness of OptiX' naive, greedy approach was so exceptional in contrast to ORX'. However, it is noteworthy that OptiX used 22.32 seconds to solve the allocation problem of ts_3. This is presumably due to this test set being more fluctuating than the former two, and that a clear pattern in the data is absent. As Table 4 shows, a confounding number of 81 iterations were performed, rather than the 2 and 9 for ts_1 and ts_2 respectively. In the initial allocation of this test set, OptiX' T_e was 790 seconds as opposed to 739 after the iterations, so the time used on calculating was regained with interest.

7.1.3 Test Data Collection 2

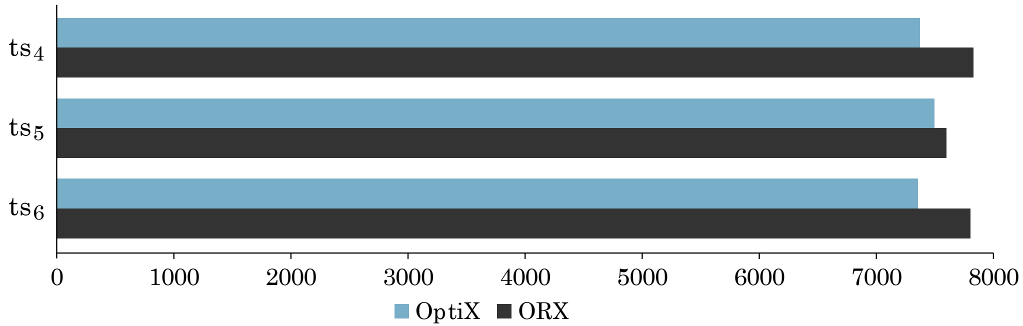


Figure 16: Results from Test Data Collection 2 in Experimental Evaluation

	Diff. from ORX	% of ORX	Iterations	Initial T_e
ts_4	456.35s	94.17%	1	7371s
ts_5	103.23s	98.64%	2	7524s
ts_6	448.10s	94.26%	10	7371s

Table 5: Details from Collection 2 in Experimental Evaluation

The second collection of test sets consisted of ts_4, ts_5 and ts_6, with 1000 tests and 10 test machines, and was designed to measure allocation performance in situations with a large number of tests and a small number of machines.

Figure 16 shows that even though OptiX provided better T_t values than ORX for all of the test sets in the collection, the difference between the two mechanisms was less prominent for this collection than for the former. The largest difference in T_t was 456.35 seconds for ts_4, and the smallest

was 103.23 seconds for ts_5. These are both insignificant numbers relative to the calculated T_t, which is more than two hours for both mechanisms on each test set. Even so, the difference in T_t between the two mechanisms for ts_4 and ts_6 were approximately the same as for ts_1 and ts_3 of Collection 1. OptiX' T_s far surpassed ORX', however, as OptiX used less than 0.2 seconds on each test set. This is a notable achievement compared to ORX, which again timed out after 30 seconds for all test sets.

7.1.4 Test Data Collection 3

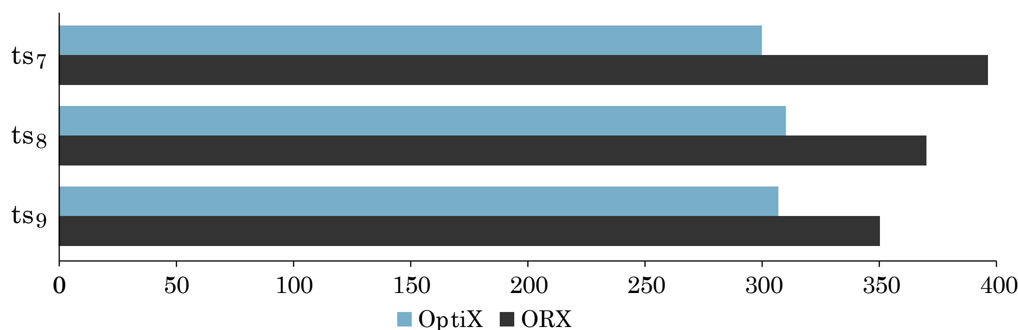


Figure 17: Results from Test Data Collection 3 in Experimental Evaluation

	Diff. from ORX	**% of ORX**	**Iterations**	**Initial T_e**
ts_7	96.19s	75.67%	3	300s
ts_8	58.91s	84.00%	8	332s
ts_9	43.19s	87.64%	21	359s

Table 6: Details from Collection 3 in Experimental Evaluation

The third collection of test sets consisted of ts_7, ts_8 and ts_9, with 200 tests and 50 test machines, and was designed to test the mechanisms in situations with more realistic number of tests, and a fair number of machines.

Figure 17 shows that OptiX again provided better total results for all test sets. OptiX' longest searching time took 1.17 seconds, while ORX once again was timed out after 30 seconds. The range of durations for the test sets in Collection 3 was minuscule compared to that of Collection 2. Accordingly, the difference between the results obtained by OptiX and those obtained by ORX represented a larger difference in percentage even though te differences were a lot smaller.

7.1.5 Test Data Collection 4

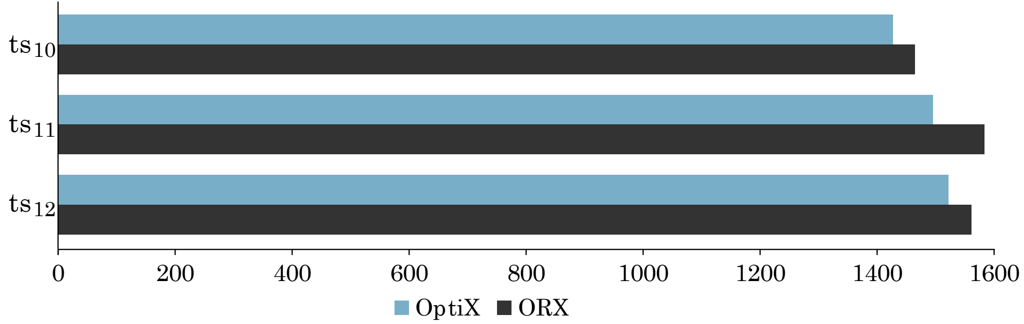


Figure 18: Results from Test Data Collection 4 in Experimental Evaluation

	Diff. from ORX	% of ORX	Iterations	Initial T_e
ts_{10}	38.04s	97.40%	2	1427s
ts_{11}	87.45s	94.48%	4	1522s
ts_{12}	39.27s	97.48%	8	1538s

Table 7: Details from Collection 4 in Experimental Evaluation

The fourth and last collection of test sets consisted of ts_{10}, ts_{11} and ts_{12}, with 200 tests and 10 test machines. This test set was designed to test the mechanisms in situations with realistic numbers of both tests and machines in terms of the context in which OptiX will be used at Altibox within the forseeable future.

Again, OptiX produced better results for all of the test sets, which can be seen in Figure 18. Because of a limited amount of possible solutions to the allocation problems, as the number of test machines was small and the number of tests moderate, the differences between the results provided by OptiX and ORX were minor.

7.1.6 Conclusion of Experimental Evaluation

OptiX provided favorable results compared to ORX for all of the test sets in the experimental evaluation. The objective of the optimization problem was to minimize the total time T_t, which is the combined time used for searching for an optimal solution and for executing the tests. Not only did OptiX provide better T_t values for the whole group of test data; *both* the

searching time T_s and the execution time T_e were superior for every test set. However, the difference in performance were small for Collection 4, which ultimately mimics the most realistic scenario.

OptiX excelled especially in two areas: in general searching time and in finding outstanding solutions to allocation problems with large numbers of both tests and test machines, and thus a great deal of possible allocations.

The searching times used by OptiX were generally exceptional, aside from for ts_3, where 22.32 seconds were needed to solve the problem.

Judging from the results obtained in the experimental evaluation, OptiX consistently used the more time to solve the last test set in each collection than the remaining. Whereas the two first test sets in each collection were allocated in virtually no time, and using few iterations, a larger number of iterations and a greater amount of time were required to allocate the last test set in each collection. This is likely linked to the tests in these test sets being executable on an arbitrary number of machines, and the test sets thus being more irregular without conforming to a pattern, as opposed to the remaining test sets.

It is a clear trend that ORX requires a lot of time to find good results, and that the quality of the final results is poor compared those of OptiX, for the most part. Nonetheless, the allocations made by ORX formp Collections 2 and 4 were certainly feasible.

In conclusion, OR-tools may be an excellent library for solving constraint programming and combinatorial problems, but OptiX provided better results for this specific problem. The main contributing factor to this is presumably that OptiX was designed and implemented specifically to solve this exact type of problem in an efficient way, whereas OR-tools was designed to provide feasible solutions to a much broader range of problems. Mechanisms built on OR-tools are therefore not guaranteed to make the best decisions at any point in the process, so the time needed to identify good solutions is inclined to take more time. This is likely the reason why ORX did not stop before the timeout of 30 seconds on any of the test sets, whereas OptiX used less than a second on most of them.

7.2 Threats to Validity

Judging from the results obtained in Section 7.1, it is clear that OptiX performed superiorly compared to ORX in the experiments. It is therefore important to establish some possible contributing factors that may threaten

the validity of the results.

As explained in the previous section, the test data was randomly generated. This means that there might have been some degree of chance involved, and that if the test sets were generated again, other results may be obtained. Additionally, there might have been some interesting situations in which to test the performance of the allocation mechanisms, that are not covered in the experimental evaluation. Even though the author tried her best to come up with realistic and representative scenarios to cover the allocation mechanism in full measure, there might have been some relevant scenarios that were not tested. Also, there might be some scenarios that would provide value despite being deemed pointless to test by the author.

The results may also depend on the performance of the machine that the experiments were conducted on. Had the experiments been conducted on a newer and faster machine with more available resources or a different operating system, the results would likely come out slightly different. ORX would perhaps be able to reach better solutions before being timed out, and could thus possibly be a stronger competitor, whereas the results obtained by OptiX would not likely be affected as much, as most of the problems were solved in less than a second.

Another contributing factor may be the author's knowledge of and experience with OR-tools being far from complete. With limited documentation and general online and literary coverage, exploring every corner of the library just could not be done with a narrow time-frame and other tasks that had to be prioritized. Although the author did invest a fair amount of time and energy in getting acquainted with the tools and tried her best to implement ORX to be as strong of a competitor to OptiX as possible, it may be a very real possibility that there are ways to implement it that would provide greater efficiency. Another possibility is that there are other optimization libraries that could potentially accomplish better results in this optimization problem than OR-tools.

7.3 Discussion

OptiRun is especially well-suited for minimizing the overall execution time of large test sets in a distributed environment. The OptiX allocation mechanism attempts to allocate test to machines in the system in such a way that the overall execution time plus the time used to search for this allocation is minimized, which represented a major objective in this project. The results

obtained and discussed in Section 7.1 shows that this goal has essentially been successfully achieved. OptiX consists of three steps: preliminary sorting, initial allocation and improvement iterations. The initial allocation is a greedy algorithm which uses a slightly altered LJF approach. With respect to the goal that was to minimize the overall execution time, this approach provides flexibility toward the end of the allocation. Had the goal been to maximize the throughput in terms of number of tests per time unit, or to minimize the average response time, a SJF approach would perhaps have been preferable.

A downside to the works of the test execution in practice is that test allocations are predetermined and final. If, for example, one test machine is particularly slow, this machine could potentially require a great deal more time to finish executing its allocated tests than the remaining machines, without the possibility to hand over tests to other machines. As test sets are handled in batches, and a new batch will not be handled until every test in the previous batch has finished executing, the execution of a whole batch of tests could thus be greatly delayed due to this one particularly slow machine.

A different problem is that a browser can crash or stop responding to the Selenium Standalone Server. This is an issue that occurs from time to time, and especially frequently with Internet Explorer, in the author's experience. If these events were to happen, the test script would fail even though the functionality might be correct. Thus, any problems linked to Selenium Grid may directly affect OptiRun by extension.

Yet another noteworthy aspect is that different browsers behave differently, which means that they do not necessarily respond identically to the browser commands in test script. This issue is out of OptiRun's control, as it related to the how the different browsers work.

Because a distributed environment constitutes an essential element of OptiRun, a requirement for the system to work is that it runs in a network without limiting restrictions. First of all, TCP/IP communication must be permitted, and the ports used for the hub and the nodes in the grid must be open and available. As must the ports used for communication between the webserver and the controller. Further, HTTP traffic must be allowed to enable the webserver to be reached from more machines than the one on which the it is running. The ports used in OptiRun are listed in Table 8.

On a general note, there are a lot of factors that can lead to system failure. Distributed environments may involve a number of potential error-

Used By	Port	Protocol
Selenium Hub	4444	TCP/IP
Selenium Node	5555	TCP/IP
OptiRun Webserver	80	HTTP
OptiRun Controller Schedule Listener	5006	TCP/IP
OptiRun Controller Request Listener	5005	TCP/IP

Table 8: Ports Used in OptiRun

prone areas because of the amount of elements involved. Apart from network issues, other potential areas of failure for OptiRun are, to mention a few: aforementioned issues related to Selenium, unstable test machines, browser versions incompatible with the Selenium drivers, and in general operating system and browser related errors. Implementing error handling of, and workarounds for, all possible areas of failure is a daunting task that have not been prioritized during the work on this project.

7.4 Return on Investment

Altibox has great interest in how adopting OptiRun in their testing process can affect their company from a business perspective. This section aims to highlight some of these aspects.

All of the frameworks used in the development of OptiRun are open-source. It has therefore not been necessary to pay for any licenses. OptiRun itself is also handed to Altibox with no charges, so the associated investment solely consists of time and resources contributed by the author and her supervisors.

A priority during the design phase has been to create a user-friendly, intuitive and consistent interface. Building the website on the Django framework has rendered this an easy task. Getting to know the interface and learning to use it is thus not expected to require excessive amounts of time and effort. Also, a brief user manual is included both on the website and in Appendix C of this thesis.

As with test automation in general, writing test scripts for OptiRun requires some technical knowledge. Being acquainted with the Python programming language and the Selenium libraries is a necessity. Writing these scripts will also take time, and additional time associated with maintaining said scripts should be expected. Furthermore, at least one computer, preferably more, should be available solely to be used for OptiRun.

The success of applying OptiRun partially depends on how the product is used. As explained in Chapter 2, caution should be used upon determining the automation coverage and which exact test should be automated. The media content in TV Overalt is dynamic, as new movies and TV shows are released and made available, while old content is removed after a while. Thus, content-based tests are not likely to be durable, and will presumably require regular maintenance.

But there are a lot of benefits to OptiRun too, if used correctly. One of these benefits is significantly extended test coverage. Currently, TV Overalt is available in Chrome, Edge, Firefox, Internet Explorer and Safari. During an acceptance test, the test team often has limited amount of time, which means that they will not be able to perform all of the tests in all browsers, and have to select which browsers should be tested. This will most likely no longer be a problem after incorporating OptiRun, as each test script can be executed in all of these browsers. Once the test scripts are written and in working order, they can be executed rapidly, precisely and repeatedly with no additional expenses and virtually no time used by human resources, and in all of the aforementioned browsers, except Safari, with only a few keystrokes. The remaining time can thus be used for manually conducting the tests unsuitable for automation. This way, it will now be possible to cover all of the supported browsers even with a limited time frame, and thus be able to test the test object more thoroughly and detect more defects.

Safari and general Macintosh support is excluded from OptiRun due to the author's lack of access to such a machine during her work on this project. Support for this platform is suggested as further work in Section 8.1. An advantage of OptiRun is that it can be easily extended.

Detecting more defects will likely result in improved product quality, and by extension additional time to test the product to an even greater extent. Additional effects may be fewer customer inquiries and higher customer satisfaction, which could lead to increased customer confidence in the product as well as in the company.

Another strength to OptiRun is that even though developing test scripts is a task that requires technical staff, once the scripts are written and uploaded, no technical background is needed to execute the tests, which would be the case without the user interface of OptiRun.

As explained in the previous section, writing Selenium tests generally does not require advanced programming skills and is not immensely time-consuming. The code involved is fairly simple, as it is mostly compromised

of locating web elements based on the class or id names, or similar, of the elements in the HTML code, and clicking buttons or filling out text fields. The time used for test script coding and maintenance will likely take approximately the same amount of time as a few manual executions of the same test, and thus provide great value in the long term.

As opposed to some other cloud testing services which execute tests on remote machines in distant locations, OptiRun can execute tests on un-deployed versions of the test object, that are only available on the network that the OptiRun server is connected to. In practice, this means that the tests uploaded to OptiRun can be executed on the test object before it is deployed, and thus be able to identify defects earlier.

OptiRun provides a JIRA integration that will enable the testers to save a lot of time on reporting failed test executions to the issue tracking system, which will be of great value. Locating issues linked to a specific test can also be done without having to open the JIRA website and search for issues with specific attribute values that comply with the specific test.

8 Further Work

The Altibox testing staff are planning to adopt test automation by introducing OptiRun in the testing of TV Overalt, and have already started discussing possible extensions. The tool provides a decent foundation for embarking on test automation, but is in its current state by no means perfect. This chapter presents some suggestions for further work that could improve the value of the product in order to further satisfy Altibox' specific needs as well as increasing the worth of OptiRun for more general use.

8.1 Extended Browser & Platform Support

Since the author did not have access to a Macintosh computer during the work on this project, OptiRun does not support this operating system, and by extension, it is also not possible to run tests in the Safari browser with OptiRun. Additionally, mobile devices are not supported as test machines in this project, so OptiRun can not be executed in browsers on such devices. All of the aforementioned browsers and platforms are compatible with Selenium, and could without further ado be included in a Selenium Grid, and therefore in OptiRun.

Safari on Mac machines and mobile Apple devices as well as Chrome on mobile devices running the Android operating system are all supported by TV Overalt, which means that Altibox must include these browsers and

platforms in the testing process. By extending OptiRun to also support these, the automation coverage of TV Overalt could be greatly increased.

8.2 App Testing

Appium is an open source, cross-platform test automation framework for use with both native and hybrid mobile apps as well as mobile web apps [30]. It supports iOS, Android and FirefoxOS, and is compatible with Selenium Grid [31]. Including Appium in the project would extend OptiRun to also support testing of apps for mobile devices.

For Altibox, who also has app versions of TV Overalt for Android and iOS, this would mean that TV Overalt could have some of its testing covered by automation in all of its forms. Since the apps looks and behaves slightly different on different screen sizes and OS versions, the apps are currently manually tested on a large selection of devices. This involves a great deal of work being repeated, which could be avoided if OptiRun were extended to include app testing as well.

8.3 Notifications

Scheduling tests for future execution and then being able to forget about them and not having to go back to the execution log to see the result, while still being notified if something went wrong, would be highly convenient. This is especially true for tests that are scheduled to be executed repeatedly or if OptiRun were to be adopted in part as a monitoring service.

This could become reality by implementing a notification service in which OptiRun on one end sent out a message upon a failed test execution, and an application on the other end received the message and created a notification based on the content. One way of doing this is to use *Google Cloud Messaging* (GCM) [32], or its successor, *Firebase Cloud Messaging* (FCM), [33], which are both cross-platform messaging services created to deliver messages for notification purposes. They support iOS and Android as well as the Chrome web browser, where it can be used for browser extension notifications. This improvement requires app development, but FCM provides instructions for incorporating the messaging solution on their website, so implementing minimal apps made solely for notifications should not require too much effort. Nevertheless it is also a possibility to invest more time and effort in the mobile app development, and create apps with some of the same functionality as the web-based user interface of OptiRun.

8.4 Continuous Integration

Test automation is commonly applied to continuous integration (CI) [7] environments, which automatically builds and tests code contained in specified repositories. It can provide frequent and rapid feedback on the code in uploaded commits, and can thus be of excellent value for software developers.

Jenkins, a widely used CI tool, offers a Selenium plugin [34] that turns a prepared Jenkins cluster into a Selenium Grid cluster, and thus allows for execution of Selenium tests in the Jenkins cluster. This way, executions of Selenium tests uploaded to OptiRun could be automatically triggered by for instance GitHub commits. This way, the developers could get instant feedback on their committed code, which could lead increased cost-efficiency as bugs could be identified as early as possible.

9 Conclusion

This thesis presented OptiRun; a platform for optimized execution of automated web tests in distributed environments. OptiRun consists of a two main parts. The controller is designed to manage the distributed system as well as to allocate and execute tests, and then report the test results. The web-based user interface was created as a means to operate the system. Its intended use is to manage test scripts, to request immediate or planned test executions, and to view results from previous executions. An integration with the issue tracking system JIRA is also included, to allow for effortless reporting of failed tests. OptiRun is written in Python, and builds on frameworks such as Selenium and Django.

OptiX, a mechanism intended for strategically allocating tests to machines in the distributed environment, was designed and implemented as part of the thesis. The ambition of OptiX was to minimize the combined time needed to search for an optimal solution and to perform the test executions of a test set. An alternative allocation mechanism was implemented for benchmarking purposes to better evaluate the performance of OptiX. The alternative mechanism was built on OR-tools, and was named ORX.

An extensive experimental evaluation was performed, where both mechanisms were tested against 12 different test sets, with the aim of evaluating their performances in situations with varying numbers of tests and machines. OptiX provided excellent results, and surpassed the benchmark values for

every test set. Judging from the results obtained in the experimental evaluation, OptiX performs especially well for problems with large sets of both test and machines, and thus a broad range of possible solutions to the allocation problem. OptiX also attained superior achievements in regard to solving allocation problems efficiently.

OptiRun was created to support Altibox in the procedure of effectively incorporating test automation as a practice in the testing process of their online web service TV Overalt.

Appendices

Attachments

A.1 Program Files

The complete program files involved in this project are included in the attachments of this document.

A.2 Printer-Friendly Version of Thesis Report

A version of this thesis report which includes some additional blank pages to make the document suitable for printing is included in the attachments of this document. Apart from the blank pages, the number of pages specified on the cover page and the attachments (excluded from the print-friendly version), the two documents are identical.

B
Setup Instructions

Because of network requirements, OptiRun will not work on restrictive networks such as *eduroam*. It will, however work on the *unix* network at the University of Stavanger.

OptiRun supports the following platforms:
Server: Windows
Test Machine: Windows & Linux

Note that since there are a lot of steps required to set up OptiRun, and the setup process therefore may be error-prone, 3 virtual machines with the system already set up was submitted on a USB memory stick along with this thesis.

B.1 Windows

The following steps are required for setup of OptiRun on Windows:

1. Download and install Python 2.7.11. `https://www.python.org/downloads/release/python-2711/`
2. Add Python to path:

(a) Open a file explorer, right click on **This PC** and select **Properties**

(b) Click **Advanced system settings** and open the **Advanced** tab

(c) Click **Environment variables...**

(d) Double click the **Path** variable under **System variables**

(e) Add the following to the variable value:
C:\Python27\;
C:\Python27\Scripts\;

3. Install PIP by entering *python get-pip.py* in the terminal. The script can be found here at `https://bootstrap.pypa.io/get-pip.py`.
4. Download and install Java. `https://java.com/en/download/`
5. To set up an OptiRun server, proceed to B.1.1, and to set up an OptiRun test machine, proceed to B.1.2.

B.1.1 Server

The BAT script called **OptiRun server** (located in the OptiRun project folder) should take care of all necessary package installations, and start OptiRun, but if something goes wrong while executing the script, the following packages must be installed/updated using the **pip install -U** command:

```
django
selenium
jira
virtualenv
virtualenvwrapper
python-dateutil
beautifulsoup4
urllib3
```

The Python script **start.py** must then be executed.

This will open 2 separate terminal windows; one for the webserver and one for the controller. The webserver is ready when the following output appears:

```
1 Performing system checks...
2
3 System check identified no issues (0 silenced).
4 June 13, 2016 - 14:55:18
5 Django version 1.9.7, using settings 'dashboard.settings'
6 Starting development server at http://<IP>:80/
7 Quit the server with CTRL-BREAK.
```

The URL on line 6 can then be entered in a browser to open the OptiRun user interface. (username: admin, password: theadmin)

The controller is ready with the following output appears:

```
1 OptiRun Controller starting up...
2 Zipping Test Machine package for Linux...    Done.
3 Zipping Test Machine package for Windows...  Done.
4 Listening for execution requests...
5 Listening for schedule updates...
6 Selenium Grid hub is up and running.
```

B.1.2 Test Machine

When the server is set up, open a browser and enter the URL specified in B.1.1. Log in (username: admin, password: theadmin), and go to the *Download* page. Click **Download OptiRun Test Machine package for Windows**. Unzip the downloaded archive file and run the BAT script called **OptiRun Test Machine**. This should take care of installing the Selenium package and start an the OptiRun test machine script, but if something goes wrong while executing the script, the Selenium package must be installed/updated using the **pip install -U selenium** command. The Python script **start.py** must then be executed.

Remember to approve the machine in the OptiRun user interface before use.

To run tests in Internet Explorer, Protected Mode must be set to the same value (enabled or disabled) for all zones. Click **Internet Options** from Internet Explorer's settings menu, select the **Security** tab, and either check or uncheck **Enable Protected Mode** for all zones.

B.2 Linux Test Machine

After having set up an OptiRun server by following the instructions in B.1 and then B.1.1, Linux machines can be set up as OptiRun test machines.

When the server is set up, open a browser and enter the URL specified in B.1.1. Log in (username: admin, password: theadmin), and go to the *Download* page. Click **Download OptiRunTest Machine package for Linux**. Unzip the downloaded archive file and run the BAT script called **OptiRun Test Machine**. This should take care of installing the required packages and start an the OptiRun test machine script, but if something goes wrong while executing the script, the following packages must be installed/updated using the **sudo apt-get install --upgrade** command:

```
python
python2.7
java-common
default-jre
selenium
```

The ChromeDriver must be made executable. Enter **sudo chmod +x drivers/chromedriver** in the terminal.

The Python script **start.py** must then be executed.

Remember to approve the machine in the OptiRun user interface before use.

B.3 OR-Tools

To be able to run ORX, OR-tools must be installed. Installation instructions can be found here: `https://developers.google.com/optimization/installing#python`.

C
User Manual

C.1 Writing and Uploading Test Scripts

1. Test scripts used with OptiRun should strictly conform to the test script template found in the **Downloads** page. Execute the script locally before uploading, to verify that it works.
2. Go to the **Test Case** page and click **Add Test Case**.
3. Give the test a suitable name, upload the test script and provide a description (optional).
4. Test cases can be added to groups.

C.2 Executing Tests

1. Go to the **Test Case** page. Mark the test cases you want to execute, and select **Execute Now** from the **Actions** menu.
2. Specify the environment you wish the tests should be executed in. You can select multiple browsers, but only one operating system for each test. Optionally, you can select the 🔀 symbol to allow the test to be executed in a random browser and operating system based on the available test machines.
3. Click **Execute**.

C.3 Scheduling Tests

1. Go to the **Schedule** page and click **Add Schedule**.
2. Give she Schedule a suitable title and set a start time. This will be the time of the first schedules execution.
3. Check the **Repeat** checkbox if you wish for the schedule to repeat regularly. Specify recurrence pattern and whether the schedule should continue indefinitely or until a specified date.
4. Add any test groups or individual tests you wish should be included in the schedule.
5. Schedule items are activated upon creation, but can be deactivated from the **Actions** menu. Only activated will be executed as planned.

C.4 Managing Test Machines

1. To register a new test machine, go to the **Test Machine** page and click **Add Test Machine**. Enter the hostname of the test machine and hit Save.
2. Test machines can be approved or disapproved from the **Actions** menu. When a test machine connects to the system, details regarding the installed browsers and the operating system are stored.
3. Test machines that connects without being registered will automatically be disapproved. They must be approved before OptiRun can execute tests on them.

C.5 Test Results

- To view the results from test executions, go to the **Execution Log** page.
- Detailed information about each test execution can be found in the detail page of every execution.
- Failed executions can be automatically reported to JIRA from the log list. Mark the executions you want to report, and select **Report to JIRA** from the **Actions** menu. A comment saying that the bug has been reproduced will be left in any open JIRA issues on the selected tests. If there are none, a new issue will be created.

- A clickable list of JIRA issues linked to each test in the execution log, as well as their statuses, is available in the log detail page.

C.6 User Administration

- Only superusers or users with special permissions can add, edit or delete other users.
- Only users with staff status will be able to log into OptiRun
- To register a user, go to the **User** page and click **Add User**. Enter the username and a password, and hit **Save**. This will lead you to a page where more information can be added, including active, staff and superuser status and user permissions. The password can later be changed by the user.
- All passwords are encrypted with a strong encryption algorithm, and it will not be possible to extract passwords from the database.
- The content of the OptiRun web interface changes according to the permissions of the logged in user.

C.7 Troubleshooting Test Machines

- Check that Java installed.
- Check that Python 2.7.x is installed.
- The OptiRun server may have been moved to a new location since you downloaded your Test Machine package. Open the OptiRun user interface and download a new package.
- Your Selenium installation might be out of date. Open a command prompt and enter **pip install -U selenium** (Windows) / **sudo apt-get install –upgrade selenium** (Linux).
- WebDriver files may not match your installed browsers. The drivers are located in the **drivers/** directory of the archive file, and can be replaced if necessary (but do not change their names).
- **Linux**: The ChromeDriver must be made executable. Enter **sudo chmod +x drivers/chromedriver** in the terminal.
- **Windows**: To run tests in Internet Explorer, Protected Mode must be set to the same value (enabled or disabled) for all zones. Click

Internet Options from Internet Explorer's settings menu, select the **Security** tab, and either check or uncheck **Enable Protected Mode** for all zones.

D
Poster

The poster on the following page was presented on the annual poster presentation for master's theses in computer science and electrical engineering at the University of Stavanger.

OptiRun

A Platform for Optimized Test Execution in Distributed Environments

MASTER'S THESIS | JANICKE FALCH | JUNE 2016

INTRODUCTION

Our society is becoming increasingly dependent on computers and digital media. The importance of, and demand for, high-quality software has become substantial. Pressure on software vendors to deliver frequent releases of quality software requires efficiency in every stage of the development process. Software testing is an important part of this process, as it serves to provide quality assurance and defect detection as well as ensuring that the test object meets its requirements. With test automation, software testing can be performed rapidly, precisely and repeatedly.

The telecommunications company *Altibox AS* has long wished to incorporate user-level test automation in the testing process of their web application *TV Overalt*, but has failed to make it a priority until now.

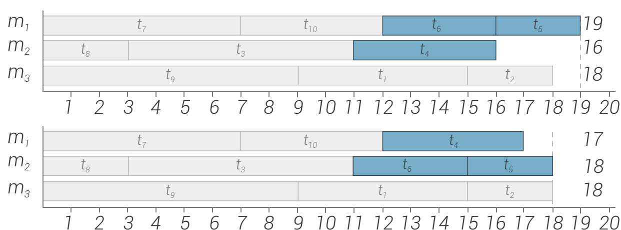
This thesis presents *OptiRun*; a platform where Altibox can run parallel tests in a distributed system. OptiRun is operated from a web-based interface that was developed as part of the thesis. A major objective has been to design and implement *OptiX*; a mechanism for allocating tests to machines in such a way that the overall execution time of a test set is attempted minimized.

METHODOLOGY

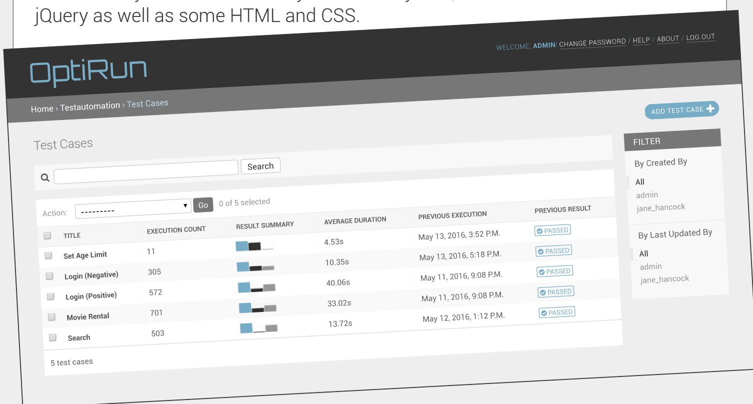
OptiRun consists of two main elements; a controller, which takes care of test allocation, execution and result reporting, and a web application where users can upload and manage test scripts, request test executions, view execution results and report failed test executions to the issue tracking system *JIRA*.

This project was written in the simple, but powerful high-level programming language Python. *Selenium*, a software testing framework for web applications, was used for test execution, and the accompanying *Selenium Grid* was incorporated to enable remote execution in distributed environments.

In order to minimize the overall execution time of a test set where the tests have resource constraints determining which machines they can be executed on, the tests must be carefully allocated to machines in the distributed system. An allocation mechanism which has been named OptiX takes care of this. The tests are first strategically sorted before being allocated using a greedy algorithm. After the initial allocation, OptiX attempts to improve the result by identifying two subsets of tests currently allocated to two different machines, that when swapped will reduce the overall execution time. This improvement step is repeated until OptiX can no longer find an improvement, or it times out. The time used to allocate the tests is also taken into consideration. The two figures below show the allocation state before and after the conduction of such an improvement in a scenario with three machines and ten tests. Note that the durations of the tests used in this example are artificially shortened compared to what they would normally be in a realistic situation.



OptiRun's web-based user interface was built on the Python Web framework *Django*, which allows for rapid development and seamless interaction with the rest of the system. It was mainly written in Python, but also includes elements of jQuery as well as some HTML and CSS.



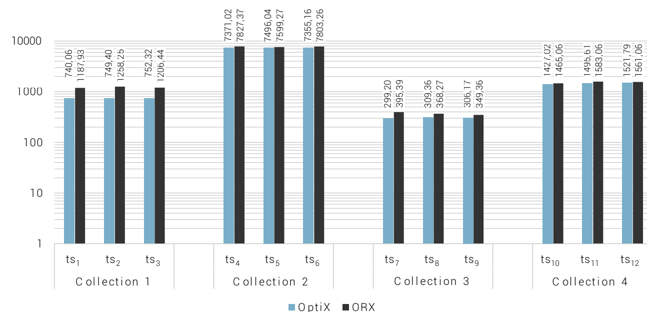
RESULTS

In order to measure and evaluate the performance of the test allocation mechanism OptiX, which represented a major objective in this project, an alternative allocation mechanism was also implemented. This was done using *OR-tools*, Google's library for combinatorial programming and constraint optimization. This alternative version was named *ORX*.

The test data used in the experimental evaluation is divided into four collections, each consisting of three test sets. The test sets in each collection all represent scenarios with a given number of tests and available machines. What separates the test sets in the same collection is the number of machines each test in the test set can be executed on. This means that there is a varying number of combinatorial solutions to the optimization problem. The test durations and the specific machines each test could be executed on were determined at random. Details about the test data is displayed in the table below:

	# OF TESTS	# OF MACHINES	# OF MACHINES TESTS ARE EXECUTABLE ON
COLLECTION 1	1000	100	ts_1: 100, ts_2: 10, ts_3: *Random*
COLLECTION 2	1000	10	ts_4: 10, ts_5: 5, ts_6: *Random*
COLLECTION 3	200	50	ts_7: 50, ts_8: 10, ts_9: *Random*
COLLECTION 4	200	10	ts_{10}:10, ts_{11}:5, ts_{12}:*Random*

All of the test sets were run with ORX to establish benchmark values, and with OptiX for comparison and evaluation. The results are visualized in the graph below, which shows the combined time used for both allocation and execution. Note that the graph uses a logarithmic scale due to major variation in numbers. As the graph shows, OptiX obtained better results than the benchmark values provided by ORX for all of the 12 test sets.



CONCLUSION

This thesis presents OptiRun; a platform for optimized test execution in distributed environments. OptiRun consists of a controller and a web-based user interface from which the tool can be operated.

OptiX, a mechanism intended for strategically allocating tests with resource constraints to machines in the distributed system, was designed and implemented as part of the thesis. The aim of OptiX is to minimize the overall execution time of test sets. ORX was created as an alternative allocation mechanism. It was built on OR-tools, and was implemented for benchmarking purposes in the evaluation process of OptiX. During the experimental evaluation, OptiX provided better results for all of the test sets.

OptiRun was created to support Altibox in the procedure of incorporating test automation as a practice in the testing process of their online web service TV Overalt.

University of Stavanger

IPTV Department
Altibox AS

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

Figures

1 The V-Model 5
2 System Architecture 17
3 Test Case Module 19
4 Intermediate Page for Immediate Test Execution Requests . 20
5 Schedule List View 21
6 Execution Log List View 22
7 Execution Log Detail View 23
8 Test Machine List View 24
9 Controller Structure 27
10 JIRA Issue Description 37
11 Sorting of Example Test Set 42
12 Initial Allocation of Example Test Set 44
13 Enhancement Iteration of Example Test Set 45
14 Complete Results from Experimental Evaluation on a Logarithmic Scale 53
15 Results from Test Data Collection 1 in Experimental Evaluation 54
16 Results from Test Data Collection 2 in Experimental Evaluation 55
17 Results from Test Data Collection 3 in Experimental Evaluation 56
18 Results from Test Data Collection 4 in Experimental Evaluation 57

Tables

1 Example Test Set 41
2 Test Data Used in Experimental Evaluation on a Logarithmic Scale 51
3 Experimental Test Results 52
4 Details from Collection 1 in Experimental Evaluation 54
5 Details from Collection 2 in Experimental Evaluation 55
6 Details from Collection 3 in Experimental Evaluation 56
7 Details from Collection 4 in Experimental Evaluation 57
8 Ports Used in OptiRun 61

Listings

1 Example Selenium Test Script 13
2 Selenium Test Script WebDriver Instantiation for Remote Execution 14
3 OR-Tools Implementation Example 15
4 Sample Shell Command for Starting Selenium Grid Hub . . 26
5 Sample Shell Command for Starting Selenium Grid Node . . 26
6 Sample Output from Selenium Standalone Server Running as a Hub 28
7 Database Communication Using Abstraction API 33
8 High-Level Directory Structure & Key Files of the OptiRun Project 34
9 Model Implementation 35
10 Implementation of Model in Administrator Interface 35
11 Issue search, commenting and creation using the JIRA REST API 37
12 Recursion Rule 38
13 Python Code for OptiX Sorting Step 41
14 ORX Multiprocessing 49

References

[1] Atlassian, *Atlassin JIRA*. URL: https://www.atlassian.com/software/jira.

[2] Spillner, A., Linz, T. and Schaefer, H., 2011, *Software Testing Foundations, 3rd Edition.*

[3] Institute of Electrical and Electronics Engineers, *1044-2009 IEEE Standard Classification for Software Anomalies,* 2010.

[4] Craig, R. D. and Jaskiel, S. P., 2002, *Systematic Software Testing.* URL: http://flylib.com/books/en/2.174.1.22/1/. Accessed: 2016-03-14.

[5] Roberts, J., 2013, *Automated Testing: End to End*, Pluralsight. URL: https://app.pluralsight.com/library/courses/automated-testing-end-to-end/table-of-contents. Accessed: 2016-02-11.

[6] Mitchell, J. L. and Black, R., 2015, *Advanced Software Testing - Vol. 3, 2nd Edition: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst.*

[7] Fowler, M., *Continuous Integration.* URL: http://martinfowler.com/articles/continuousIntegration.html. Accessed: 2016-03-13.

[8] Marriott, K. and Stuckey, P. J., 1998, *Programming with Constraints - An Introduction.*

[9] M. Mossige, A. Gotlieb, H. Meling, and M. Carlsson. *Optimal Test Execution Scheduling on Multiple Machines with Resource Constraints.* Technical report.

[10] Sauce Labs, *Sauce Labs Press Coverage.* URL: https://saucelabs.com/press-room/press-coverage/news-2, . Accessed: 2016-03-14.

[11] Sauce Labs, *Sauce Labs Features.* URL: https://saucelabs.com/features, . Accessed: 2016-03-14.

[12] H. Furubotten. *The Autograder Project: Improving software engineering skills through automated feedback on programming exercises.* Technical report.

[13] Swaroop, C. H., 2004, *A Byte of Python.* URL: http://python.swaroopch.com/. Accessed: 2016-02-12.

[14] SeleniumHQ, *Selenium – Web Browser Automation.* URL: http://www.seleniumhq.org/, .

[15] SeleniumHQ, *Selenium WebDriver – Selenium Documentation.* URL: http://www.seleniumhq.org/docs/03_webdriver.jsp, . Accessed: 2016-02-02.

[16] SeleniumHQ, *Selenium with Python – Selenium Python Bindings 2 Documentation.* URL: http://selenium-python.readthedocs.io/index.html, .

[17] Python, *25.3. unittest – Unit Testing Framework – Python 2.7.12rc1 Documentation.* URL: https://docs.python.org/2.7/library/unittest.html.

[18] *Distributed Testing with Selenium Grid.* URL: https://www.packtpub.com/sites/default/files/downloads/Distributed_Testing_with_Selenium_Grid.pdf, . Accessed: 2016-03-13.

[19] Django Software Foundation, *Django: The Web Framework for Perfectionists with Deadlines.* URL: https://www.djangoproject.com, . Accessed: 2016-01-31.

[20] Holovaty, A. and Kaplan-Moss, J., 2007, *The Definitive Guide to Django: Web Development Done Right.* URL: http://www.djangobook.com/en/2.0/index.html.

[21] Google, *OR-Tools User's Manual.* URL: https://or-tools.googlecode.com/svn/trunk/documentation/user_manual/index.html. Accessed: 2016-04-19.

[22] The Linux Information Project, *Standard Error Definition.* URL: http://www.linfo.org/standard_error.html, .

[23] The Linux Information Project, *Standard Output Definition.* URL: http://www.linfo.org/standard_output.html, .

[24] Python, *17.2. socket – Low-Level Networking Interface — Python 2.7.12rc1 Documentation.* URL: https://docs.python.org/2.7/library/socket.html.

[25] Anderson, T. and Dahlin, M., 2011*, Operating Systems: Principles and Practice, Beta Edition.*

[26] SQLite, *About SQLite.* URL: https://www.sqlite.org/about.html.

[27] Django Software Foundation, *Applications – Django documentation – Django.* URL: https://docs.djangoproject.com/en/1.9/ref/applications/, .

[28] Gustavo Niemeyer – Labix, *python-dateutil – Labix.* URL: https://labix.org/python-dateutil.

[29] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., 2009*, Introduction to Algorithms - Third Edition.*

[30] Appium, *Appium – Getting Started.* URL: http://appium.io/getting-started.html?lang=en, .

[31] Appium, *Appium – API Reference.* URL: http://appium.io/slate/en/v1.2.3/, .

[32] Google, *Google Cloud Messaging.* URL: https://developers.google.com/cloud-messaging/.

[33] Google, *Firebase Cloud Messaging.* URL: https://firebase.google.com/docs/cloud-messaging/.

[34] Jenkins, *Jenkins Wiki – Selenium Plugin.* URL: https://wiki.jenkins-ci.org/display/JENKINS/Selenium+Plugin.