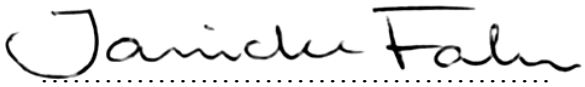
Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: *Computer Science*	Spring semester, 2016 Open / ~~Confidential~~
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 · Constrained Programming · Scheduling · Python · Selenium · Django*	Pages: …… + attachments/other: …… Stavanger, 15.06/2016 Date/year

OptiRun

A Platform for Optimized Test Execution
in Distributed Environments

altibox

Janicke Falch
June 2016

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

Abstract

TODO

Acknowledgements

TODO

Contents

1 Introduction

Software systems, ranging from business applications to consumer products, have grown to form a fundamental element 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 being increasingly 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 to the software development process. It plays a critical role in quality assurance and defect detection, and is an important means to ensure that the software in question behaves as expected and according to specifications.

1.1 Origin

The subsidiary company of the Lyse Group, Altibox, is a telecommunication service provider. They offer a selection of services including broadband, *Internet Protocol Television* (IPTV) and *Voice over Internet Protocol*, (VoIP). Since its origin, the company has grown a great deal, and at a considerably higher pace than originally anticipated. Because the company had trouble keeping up with its rapid development, they 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 for 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 portable 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 on 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. It can be applied as a testing 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 a distinguished intent of optimizing the time of test runs as well as provide control and feedback of test results. The tool is intended to be utilized during system or acceptance testing by primarily technical testers.

The main area of use is execution of Selenium test scripts for web applications. Tests should be able to be executed remotely in a controlled distributed environment. An algorithm for optimally allocating test cases on the available test machines in the distributed environment should be provided. Further, an interface for uploading, managing, executing and scheduling test scripts should be included, and logs from results of previous test runs should be available. Upon a failed test, an option to report this in the issue tracking system used by Altibox, JIRA [1], 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, and presents the dashboard from the user's perspective.

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

Chapter 6 presents and discusses some experimental results and evaluates the system.

Chapter 7 provides some suggestions for further work.

Chapter 8 concludes this thesis.

2 Background

Before moving onto the technical substance, it will be beneficial to acquire a theoretical basis on the subject of this thesis. This chapter introduces software testing and explains some essential concepts. Further, it presents constraint programming and optimization as well as some previous work related to this thesis.

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 large field, only a small selection of relevant concepts and ideas will 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 action oriented toward assessing the software with the goal of determining whether it meets the required results [**?**].

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 [**?**]. 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

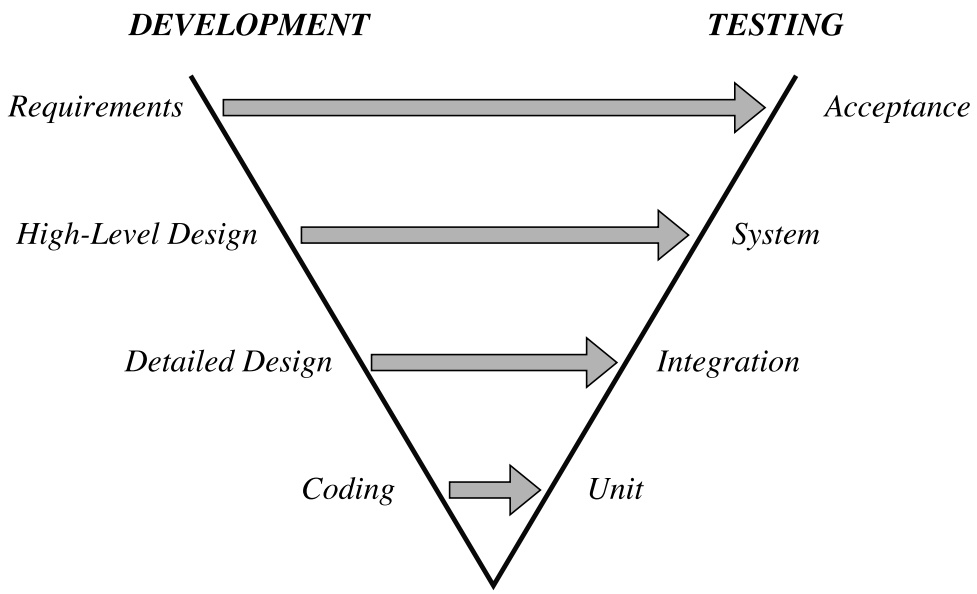


Figure 1: The V-Model

The *V-model* is an important asset in software testing, behind which the central idea is to illustrate how each development task in a software development process have a corresponding testing task of equal importance. This is symbolized by the two branches of the letter "V" in the model. The development process is represented by the left branch, which shows the system being gradually developed. The testing process is represented by the right branch, which shows how elements assembled to form 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 created using the same four levels as found in [3]. The highest development level covers the 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 can be applied to 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 unconsidered. 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 to automate web browsers. It executes functional UI tests on full system builds, and run without access or consideration to the source code and 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 does 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 at the perspective of the end user. Such tests require the system to work as a whole, and thus generally run slower and are more brittle [4], which is why functional UI tests often are the fewest.

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 [5]. 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) [6] environments. This has not been done in this project, but is presented in Subsection **??** 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 [7].

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 [7].

The optimization problem in this project does not exclusively revolve around finding an optimized allocation - in situations where there are a huge number of solutions to the problem, identifying the best solution can take a vast amount of time. For instance, in a situation with two competing solutions where one provided a 10 second longer overall execution time, but took 20 seconds less to find, this would be the preferred solution. The time it takes to arrive at the solution must therefore also be taken into account. Some stop criteria for searching are 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.

Furthermore, the following list of assumptions are expected to be strictly imposed:

Non-preemptive execution - Once a test execution has started, it will run uninterrupted until it completes.

Non-cumulative execution - 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.

Machine-independent execution time - Although the execution time will not necessarily be machine-independent in reality, depending on machine performance, this is considered an assumption.

2.3 Related Work

Some work has previously been done on the subject of test case scheduling and automation of functional UI tests for web applications. This section presents the scheduling method *TC-Sched* and the automated testing platform *Sauce Labs*.

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. Mossige *et al.* proposed the method in [8]. The authors define the *optimal test case scheduling* (OTS) as the optimization problem of finding an execution ordering and assignment of all test cases to machines, in such a way that each test case is executed once, no global resources such as measurement instruments or network devices are used by two test cases at the same time, and the overall test execution time is minimized. TC-Sched addresses situations in which some operations may require exclusive access to one or more global resources, such that only one test case can access each resource at a given time. Including the aspect of global resources in this project would have been redundant as there are no such resources involved.

2.3.2 Sauce Labs

Cloud testing services use cloud computing 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 [9], which will be introduced in Section 3.2.

Sauce Labs bear some similarities and some dissimilarities with the design of 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 [10], 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 algorithm in this project also distinguishes the two projects even further.

3
Technology

A number of distinctive 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 largely been used in the conduction of 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 [11].

Building this project on 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. Additionally, Python supports multithreading and is suitable for running a program as a service. 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. Several different tools and libraries are included in the framework, all with the

common goal of supporting browser automation. Selenium can be used to automate different types of browser jobs such as web-based administration tasks, but is primarily used for test automation. The project is released under the Apache 2.0 license and is thus free and open-sourced. Selenium has language bindings for several different programming languages, including Python. The Selenium tools used in this project will be introduced below.

3.2.1 Selenium WebDriver

Selenium WebDriver consists of a set of libraries that helps with automation of tests for web applications, and a set of executable WebDriver files, one for each browser, that perform the actual automation specified in the WebDriver scripts. Selenium WebDriver interacts directly with the browser by sending calls using the native support for automation for each browser [12]. Selenium WebDriver runs on Windows, Linux and Mac, and supports most conventional web browsers.

The WebDriver is used to create a new instance of the requested web browser. It is used to fetch a given web page, and then locates UI elements. When an element is located, the WebDriver can be used to perform an action on the element, such as clicking a button, checking a checkbox, or populating a text field.

```
1  from selenium import webdriver
2
3  # Create a new instance of the Chrome WebDriver
4  driverdriver = webdriver.Chrome("<file path>/chromedriver.exe")
5
6  # Go to the Altibox TV Overalt start page
7  driver.get("https://tvoveraltstg.altibox.no/")
8
9  # Locate and click the Login button
10 login_button = driver.find_element_by_class_name('btn-login')
11 login_button.click()
```

Listing 1: Selenium WebDriver Example

Listing 1 shows a simple example of how Selenium WebDriver can be used with Python to open the Chrome web browser, navigate to the Altibox TV Overalt start page, locate the *Login* button, and click it. OptiRun revolves around executing scripts that uses the Selenium 2.0 libraries and WebDrivers to execute automated web browser tests.

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 executing tests on multiple machines in parallel. This also opens up for running tests on different operating systems.

Reasons for wanting to incorporate Selenium Grid include being able to run tests against multiple browsers, browser versions and browsers running on different operating systems, and to reduce the execution time of the tests. In practice, a grid is made up of one Selenium Grid Server; a *hub*, and one or more slave machines; *nodes*, all running a Selenium Standalone Server. The nodes use Selenium WebDriver to communicate with the hub through a JSON wire protocol [13].

Selenium Grid 2.0 and Selenium Standalone Server 2.51.0 were used in this project.

3.3 Django

Django is a high-level web development framework that is implemented in 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 the framework as fast, secure and scalable [14]. Comparable to Selenium, Django is also essentially a collection of Python libraries [15]. The Django libraries can be imported and used to implement web applications. Some additional HTML, CSS and JavaScript code has been applied along with the Python code.

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 Chapter 4. Details concerning the implementation will be explained in Chapter 5. Version 1.9 of Django was used in this project.

3.4 OR-Tools

Google's *Operations Research Tools* (OR-tools) [16] is an open source library for combinatorial and constraint optimization. The tool set is written in C++, but is available with 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 2 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
5  variables = [solver.IntVar(0, 2) for _ in range(3)]
6
7  for i in range(len(variables) - 1):
8     solver.Add(variables[i] != variables[i + 1])
9
10 db = solver.Phase(variables, solver.CHOOSE_FIRST_UNBOUND,
       solver.ASSIGN_MAX_VALUE)
11 objective = solver.Maximize(solver.Sum(variables), 1)
12 solver.NewSearch(db, objective)
13
14 while solver.NextSolution():
15    result = [int(item.Value()) for item in variables]
16    print result, "Sum =", sum(result)
17
18 >>> [0, 2, 0] Sum = 2
19 >>> [0, 2, 1] Sum = 3
20 >>> [1, 2, 1] Sum = 4
21 >>> [2, 1, 2] Sum = 5
```

Listing 2: OR-Tool Implementation

As a means to evaluate the optimization mechanism of test allocations designed for this project, an alternative version using OR-tools has also been implemented. This alternative implementation will be presented in Chapter 5, and a discussion, evaluation and comparison of the two implementations will be presented in Chapter 6.

4
System Overview

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

4.1 Architecture

OptiRun is operated through a web-based dashboard. This web service 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. They could, however, be separated with little effort if needed.

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, test results and user authentication are also stored in the database. The web server can report defects in the issue tracker system, for which Altibox uses JIRA.

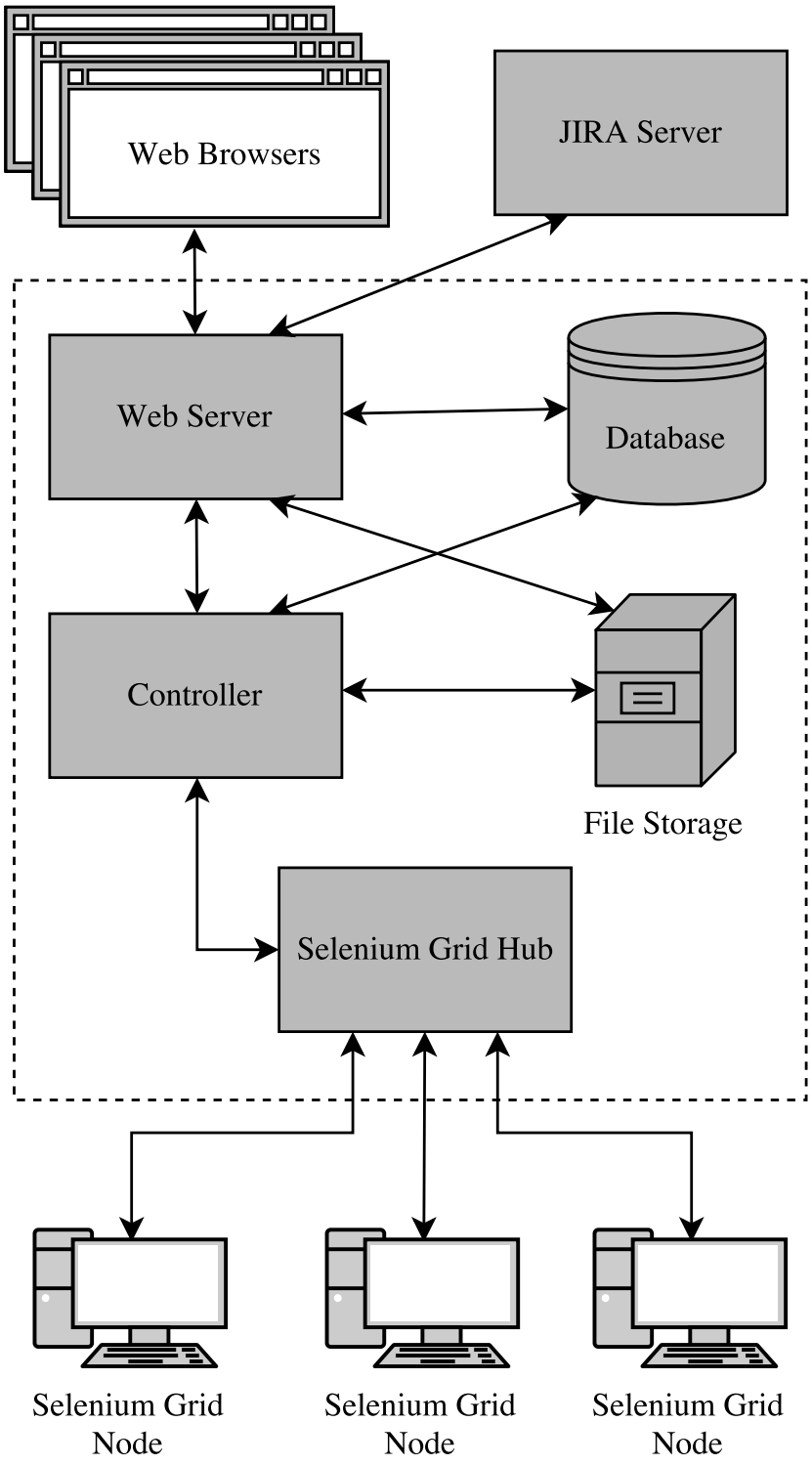


Figure 2: System Architecture

The controller listens to test execution requests triggered through the dashboard, and checks the schedule. Upon a pending execution, it fetches the given test script from the file storage. Test scripts are sent to the current instance of the Selenium Grid Hub with specifications of which node in the grid that the test script should be executed on. The hub then triggers the given node to execute the test. After the test has finished, the controller stores the result in the database.

4.2 Dashboard

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 subsection will present 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

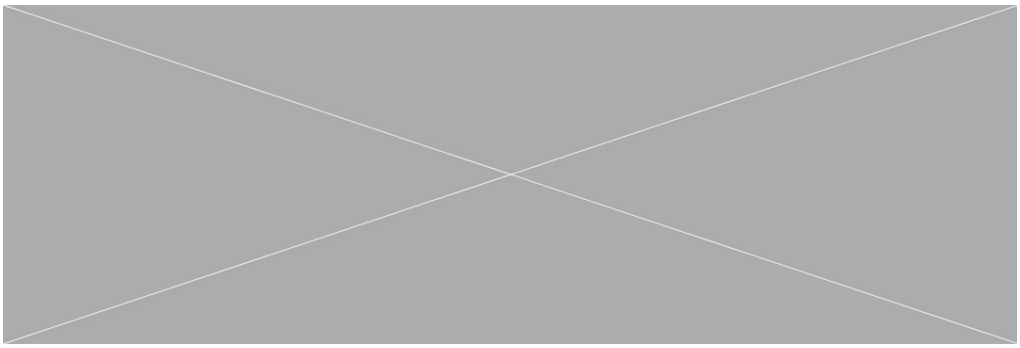


Figure 3: Home Screen

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 in the remaining subsections of this section. In order to give the user an indication of status, some graphs that show the results of recent test

runs, as well as key performance indicators and other important information is displayed on this page.

4.2.2 Authentication & Authorization

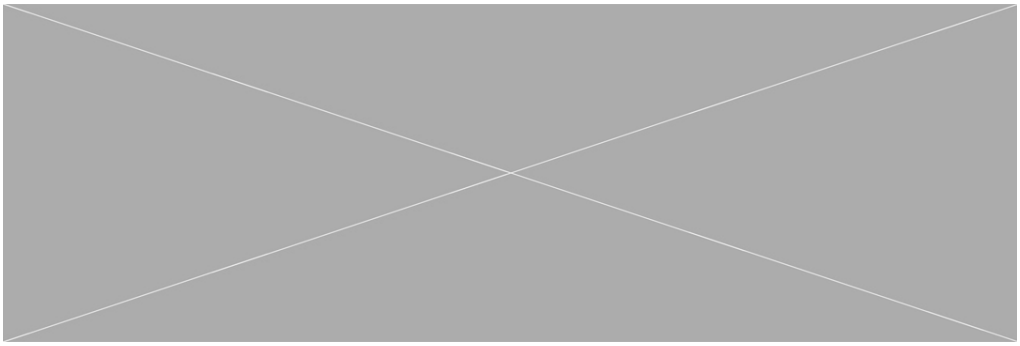


Figure 4: Authentication and Authorization Module

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 info and the permissions of other user accounts, add them to user groups, and specify the permissions of user accounts.

OptiRun is a tool that is meant to be part of a business process in which only trusted peers should be allowed to take part, and by extension, OptiRun should only be operated by trusted associates. For this reason, no user accounts can be created by anyone other than a superuser. Thus, any person who wants and account must request this by someone with superuser privileges.

By default, all passwords are encrypted by Django before being stored in the database.

4.2.3 Settings/Configuration

TODO

4.2.4 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

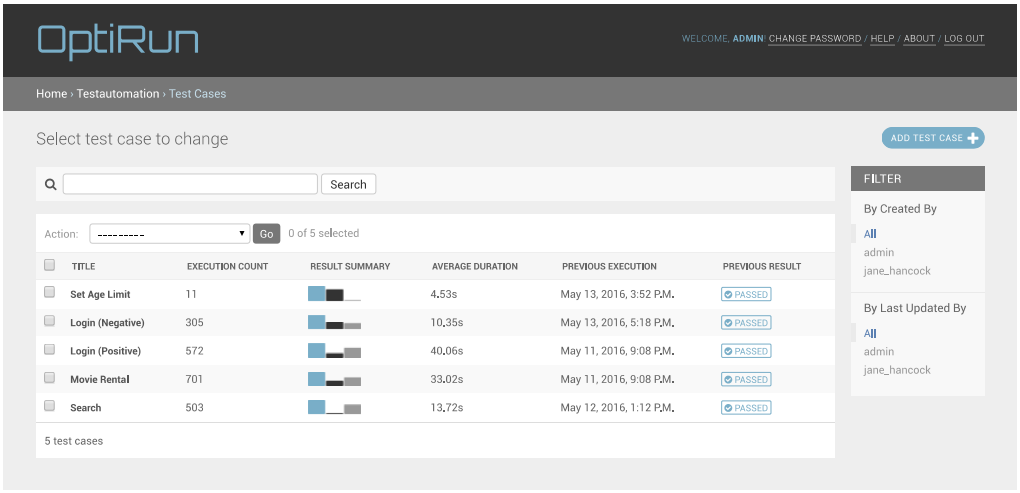


Figure 5: Test Case Module

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.

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.

It is possible to select test cases and perform actions on them. The action options - *Delete selected test cases* and *Execute selected test cases* - are located in a dropdown menu above the test case list. 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 latter 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.

When an action has been performed, or attempted performed, a feedback message telling if the action was successful or not is displayed on the screen.

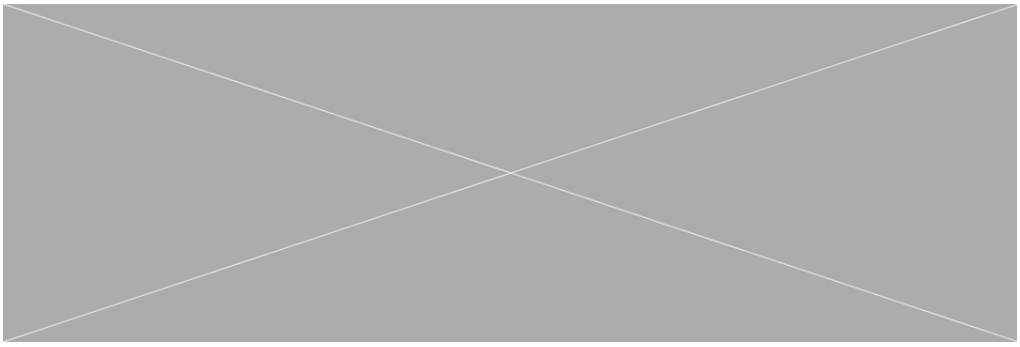


Figure 6: Intermediate Page for Immediate Test Execution Requests

4.2.5 Test Group Module

Test groups are listed similar to individual test cases as described in the previous subsection. 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 module in the test object.

4.2.6 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 creating or editing 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.

4.2.7 Execution Log

The layout of the log is the same as the remaining list views, but the functionality differs in that log items are read-only. They can not be

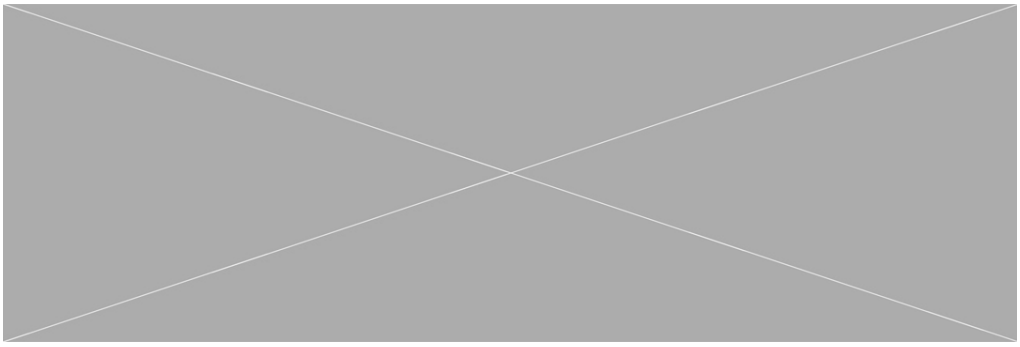


Figure 7: Schedule Creation Form

created or deleted. Log items are created and inserted into the database based on the result from test executions. Each log object corresponds to an execution of an individual test script. 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, output and of course the result.

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 displaying existing JIRA issues reported by OptiRun for the specific test, and its status.

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.

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 insight of how it works.

About This page provides a presentation what OptiRun is and how it came to be. Some licenses are also included in this page.

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 with an accompanying example to illustrate how the mechanism works step-by-step will be presented. 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

In OptiRun, Selenium Grid works as the backbone of all interaction between the server and the remaining machines in the distributed system. It is used to establish connections and to perform test executions.

Setting up a Selenium Grid environment requires all involved machines to have a *Selenium Standalone Server*, which is a file of the JAR (Java Archive) format, of the same version locally stored. The Selenium Standalone Server is started by executing a shell command.

A Selenium Grid Hub (server) can be started simply by executing the command in Listing 3, 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 can be viewed by opening `http://<hub-host>:4444/grid/console` in a browser.

```
1 java -jar "C:/selenium-server-standalone-2.51.0.jar"
```

Listing 3: 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 machines.

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 that 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.

```
1 java -jar "C:/selenium-server-standalone-2.51.0.jar" -role node -hubHost
      <hub-host> -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 -browser
      "browserName=firefox, version=45.0,
      applicationName=1b234276-fc02-11e5-b752-080027f8a664"
```

Listing 4: Sample Shell Command for Starting Selenium Grid Node

Listing 4 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://<HubHostname/`

IP>:4444/grid/api/proxy?id=http://<NodeIP>:5555.

5.2 Controller

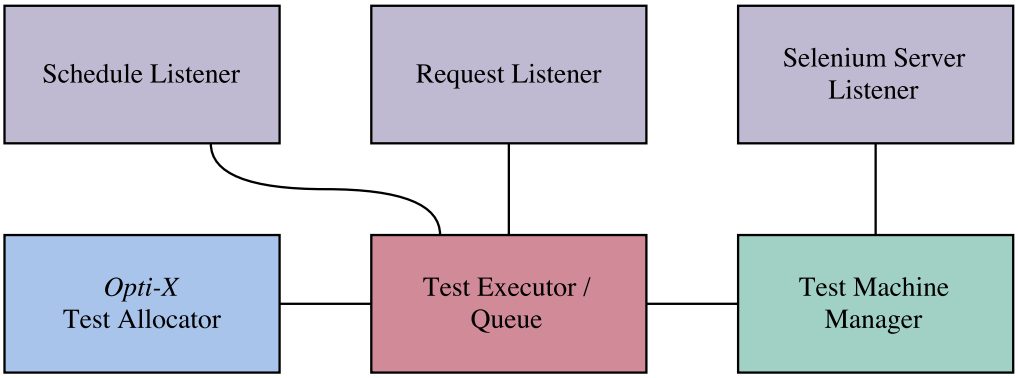


Figure 8: Controller Structure

The *controller* consists of multiple processes running in different threads. Figure 8 shows the inner structure. Each of the modules in the figure will be explained in this Section.

5.2.1 Selenium Server Listener

After starting the Selenium Grid Hub as described in Section5.1 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 and handle any incoming messages. This means that when a node connects or disconnects to the system, the Selenium Server Listener gets notified and sends notice to its instance of the Test Machine Manager.

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. As all test machines and their configuration other details are stored in the database, the any changes must be reported. To manage which

machines can be used for test executions, there is a boolean *Approved* attribute in the database. This attribute is set to *False* for test machines that connects to the grid without being registered in the database. Test machine approval and disapproval can, as mentioned in Chapter 4 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 or not they're connected to the grid. This is done by checking the HTTP response from a URL constructed by the IP address and the port number of the machine. Entering the URL of a Selenium Grid Node results in error code 403, so the test machine is only assumed live if the URL returns this error code.

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 [**?**].

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 the schedule by querying the database for schedules. 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. Like immediate test execution requests, schedule updates are also packed as JSON object and sent over the network using Python's socket library. 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.

Overriding Django's built-in functionality for model deletion is a complex task, and was thus excluded, the Schedule Listener is not notified when a schedule object is deleted. The schedule is updated every 900 seconds, or 15 minutes.

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 at the same time. 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 [**?**]. 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.

After test cases have been retrieved from the queue, allocated amongst the pool of available test machines (Section 5.3), and sorted, the execution can begin. 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 Allocation Mechanism

Since the test type intended for OptiRun generally runs slowly, designing a mechanism that would efficiently allocate tests to test machines in an attempt to minimize the duration it takes to find and execute the schedule. This mechanism has been named *OptiX*, and will be thoroughly described in the following subsection. As explained in Chapter 2, an alternative allocation mechanism has also been implemented using Google's OR-tools library, and has been used establish a benchmark for use in the evaluation process

of OptiX, which will be presented in the next chapter. The alternative allocation mechanism has been named *ORX*, and implementation details will be explained subsequently.

5.3.1 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	6	m_1, m_2, m_3
t_2	3	m_1, m_2, m_3
t_3	8	m_1, m_2, m_3
t_4	5	m_1, m_2, m_3
t_5	3	m_1, m_2, m_3
t_6	4	m_1, m_2, m_3
t_7	7	m_1
t_8	3	m_2
t_9	9	m_3
t_{10}	5	m_1, m_3

Table 1: Example Test Set

Sorting

The first step is a preparation for the initial allocation. The sorting does not play a crucial role in the allocation mechanism, but is 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. If toward the end of the allocation process there would only be left tests that could run on a certain machine, the overall execution time could be greatly affected, so the sorting avoids this by ordering them fittingly beforehand.

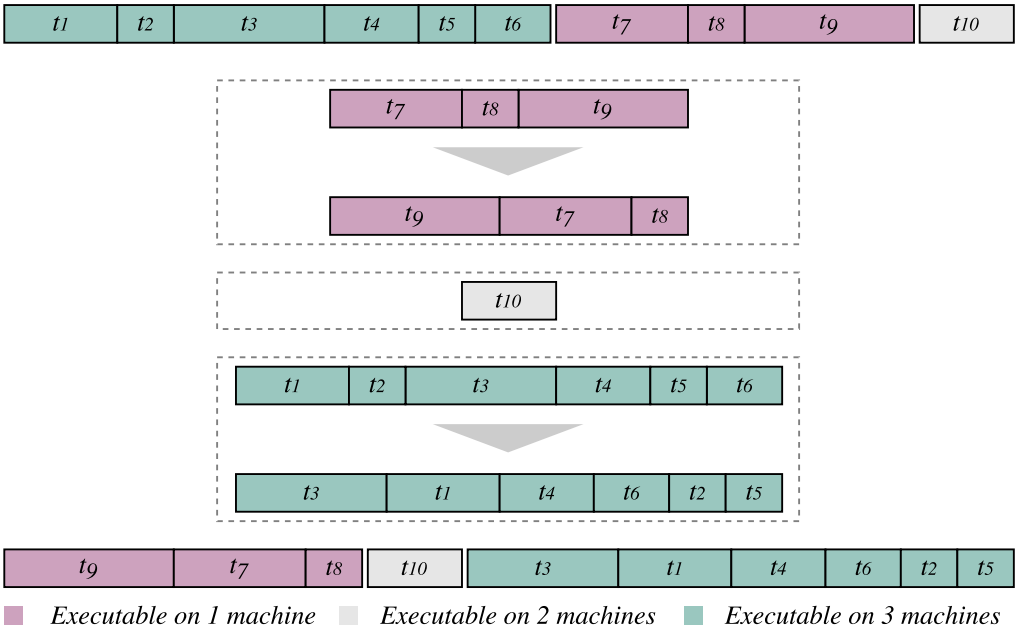


Figure 9: Sorting of Example Test Set

The test set is first divided into subsets determined by the number of machines they can be executed on. Each of the subsets is then sorted by their estimated duration in descending order, so that the longest tests are placed first and the shortest are placed last. The list is then assembled again.

Figure 9 illustrates how the sorting method 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. t_{10}, which can be executed on two machines comes after. Finally, tests t_1 through t_6, which can run on all of the machines, are ordered and added to the list of tests.

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 [17]. 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 looped through, and the tests are allocated one by one, starting with the *longest* of the tests that are executable on the *fewest* machines. Each test is initially allocated to the machine that currently has the shortest overall duration among the machines that the test is executable on.

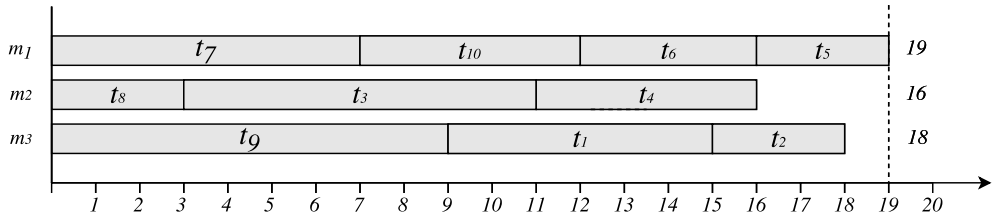


Figure 10: Initial Allocation of Example Test Set

Figure 10 shows how the test set is initially allocated among the test machines. 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. 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.

Enhancement 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 overall 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 executable on the swappee candidate are retrieved, which in this case is t_5 and t_6. A list of all possible subsets from this set is then created. This is done by

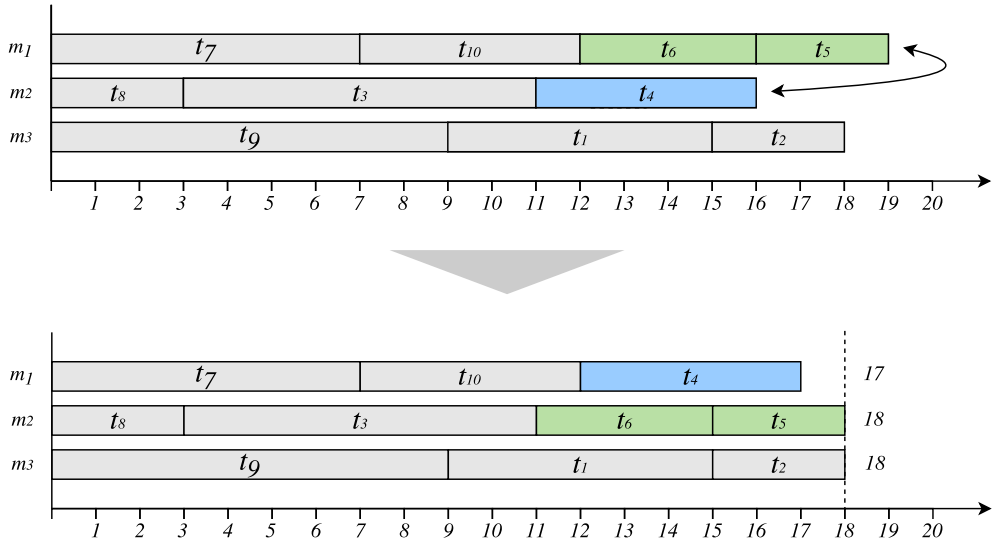


Figure 11: Enhancement Iteration of Example Test Set

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, we examine how each swap would affect the overall total 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. In this example, OptiRun found an optimal solution to the optimization problem. Figure 11 shows how the subset swap is conducted, and how it affects the overall duration.

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

ration 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 arrived at 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 decided upon:

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

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 on the problem. 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, OptiRun 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.

5.3.2 ORX

The implementation of ORX is of the same style as the OR-tools example shown in Listing 2 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 authors 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 5 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 enhancement has taken 100 times as long as it took to find the previous enhancement.
- The previous enhancement took 10 times as long to find than the enhancement itself.
- 30 seconds have passed (timeout).

```
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 solution_loop(self, durations, executable_on, shared_allocations,
        max_durations, last_updated, lock):
24      # ...
25
26      while solver.NextSolution():
27        lock.acquire()
28        # ...
29        lock.release()
```

Listing 5: ORX Multiprocessing

5.4 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 these other database systems include its being self-contained and serverless, and the database is contained in a single disk file [**?**]. For convenience concerning submission, SQLite was the preferred choice for this project. However, another SQL database engine can replace it with little effort if needed.

As explained in Chapter 3, 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 auto-generated SQL files covering creations and changes to the tables related to test automation are all placed under */testautomation/migrations/* by default. When a change has been done to one of the models, a new migration can be created by performing a few shell commands [**?**].

The database can be accessed either by writing raw SQL queries, or through Django's API for database abstraction. The former approach was

first implemented in this system, but was then changed to the latter, as it was 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 6 shows how an *insert* statement is conducted in OptiRun using this approach, although the listing is somewhat simplified as less attributes are specified. 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=1, execution_time=datetime.utcnow())
5 l.save()
```

Listing 6: Database Communication Using Abstraction API

In addition to providing excellent readability, the abstraction greatly decreases the required number of code lines needed to achieve the same result compared to executing raw SQL queries. This is because the database location does not need to be stated, connection with the database does not need to be programmatically established and then closed when the transaction is finished, etc. The abstraction takes care of all of this. Other types of queries such as *select*, *update* and *delete* are also supported with 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, such as in the schedule creation form.

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.

5.5 Dashboard

TODO: Write section intro

5.5.1 Models

With Django, data models provide the foundation on which database tables are created and maintained. A model implementation can be seen as a Python equivalent to an SQL *create table* statement. Model implementations are translated to SQL and executed by Django. Each model, which is a subclass of *django.db.models.Model*, 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.

The models are located in */testautomation/models.py*. In this file, model specifications of test cases, groups, schedules and logs are implemented.

Models can also have attributes derived from functions, which can use other attributes stored in the database. Derived attributes are used in several of the models. For instance in the test case module, there are one function that query the Log table in the database, and counts number of instances linked to the particular test case. Average duration and previous execution date are retrieved in a similar manner.

```
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
7     def times_run(self):
8        tr = # perform query
9        return tr
```

Listing 7: Model Implementation

Listing 7 shows a reduced adaption of how the test case model has been implemented. This model contains two model fields and a function. The

script field is of the type *model.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.

5.5.2 Admin

TODO: Write introduction to this subsection

Listing 8 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 7. Model administrator representations are subclasses of *django.contrib.admin.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 admin representations, a number of additional fields and specifications are also included. Custom forms validation functionality can be integrated with the creation/change form. This has been done with test case objects to ensure that only test scripts that fulfill certain criteria can be uploaded. Admin actions are also implemented here.

```
1 from django.contrib import admin
2 from .models import TestCase
3
4 class TestCaseAdmin(admin.ModelAdmin):
5    list_display = ('title', 'times_run', 'avg_dur')
6    search_fields = ['title',]
7    fields = ('title', 'script')
8
9 admin.site.register(TestCase, TestCaseAdmin)
```

Listing 8: Implementation of Model in Administrator Interface

5.5.3 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 API* (Representational State Transfer Application Program Interface) [**?**].

The JIRA integration is used for two things: reporting failed test executions and retrieving issues linked to specific test executions in the execution log. These actions require access to the JIRA server used by Altibox, which is only available when connected to Altibox' *Virtual Private Network* (VPN). To avoid problems in conjunction to VPN failure, an access check must be performed before proceeding. This is done by using the *httplib* library to attempt establishing contact with the JIRA server and setting a 1 second timeout.

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. JIRA access is checked. 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.

On the log detail page, 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 9 shows how issue search, commenting and creation are performed using the JIRA REST API. The Python code for this are located in */testautomation/admin.py*, and is called from the methods *report_to_jira* and *get_jira_issues* in the *LogAdmin* class of this file.

5.5.4 Event Recurrence

RRULE (Recurrence Rule) is a module in the Python library *dateutil*, which provides an extension to Python's *datetime* module. It is a small and fast library used in OptiRun to specify recurrence patterns of test executions. RRULE instances can be implemented multiple ways. In this project, it is achieved through passing a string with a specific format, containing

```
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('<SQL 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 9: Issue search, commenting and creation using the *JIRA REST API*

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 10.

The string in the listing above is used to create an RRULE instance in which the first occurrence is set to the 15^{th} of June 2016 at 4 pm, and repeats weekly. 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.

```
from dateutil.rrule import rrulestr
from datetime import datetime

rule_string = "DTSTART:20160615T160000\nRRULE:FREQ=WEEKLY"
rule = rrulestr(rule_string)

print rule.after(datetime.now())

>>> 2016-06-15 16:00:00
```

Listing 10: 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 Evaluation

This chapter opens with introducing a collection of test sets used to measure the performance of the allocations 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 threat 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 the investment put into this project.

6.1 Experimental Evaluation of Test Allocation

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.

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. 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

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

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.

The test sets, located in */controller/test_ data/input/*, are stored as JSON objects in files with the naming convention *test_ set_ x.json*, where x represents the number of the test set. Likewise, the results from the experimental testing are stored in files with the same naming convention, but in */controller/test_ data/output/* instead.

	OptiX			**ORX**		
Test Set	T_s	T_e	T_t	T_s	T_e	T_t
ts_1	30.04s	777.00s	**807.04s**	34.93s	1153.00s	**1187.93s**
ts_2	8.66s	752.00s	**760.66s**	34.25s	1224.00s	**1258.25s**
ts_3	0.29s	867.00s	**867.29s**	34.44s	1172.00s	**1206.44s**
ts_4	0.02s	7410.00s	**7410.02s**	30.37s	7797.00s	**7827.37s**
ts_5	0.16s	7497.00s	**7497.16s**	30.27s	7569.00s	**7599.27s**
ts_6	0.19s	7354.00s	**7354.19s**	30.26s	7773.00s	**7803.26s**
ts_7	1.72s	301.00s	**302.72s**	30.39s	365.00s	**395.39s**
ts_8	0.22s	355.00s	**355.22s**	30.27s	338.00s	**368.27s**
ts_9	0.67s	311.00s	**311.67s**	30.36s	319.00s	**349.36s**
ts_{10}	0.18s	1428.00s	**1428.18s**	30.06s	1435.00s	**1465.06s**
ts_{11}	0.60s	1520.00s	**1520.60s**	30.06s	1553.00s	**1583.06s**
ts_{12}	0.80s	1549.00s	**1549.80s**	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.

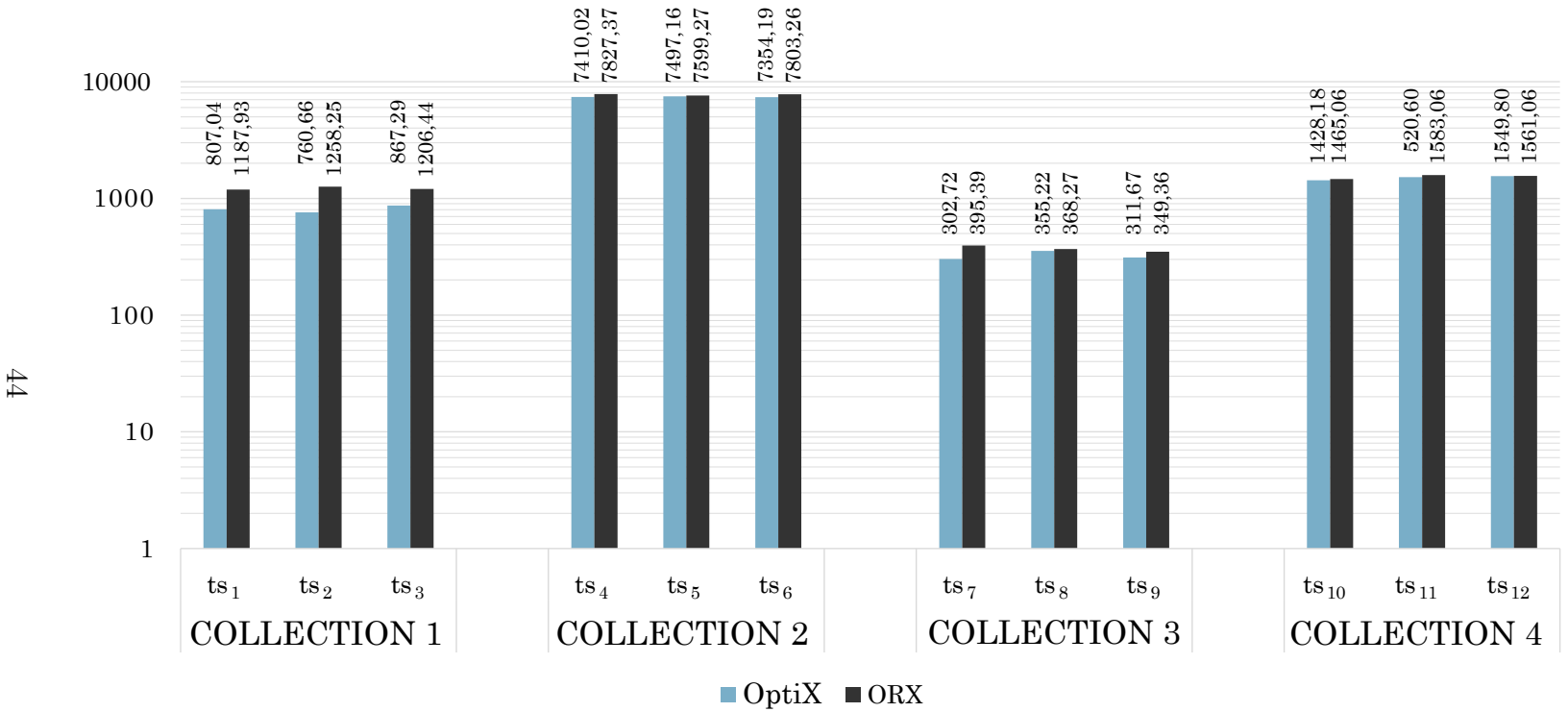


Figure 12: Complete Results from Experimental Evaluation

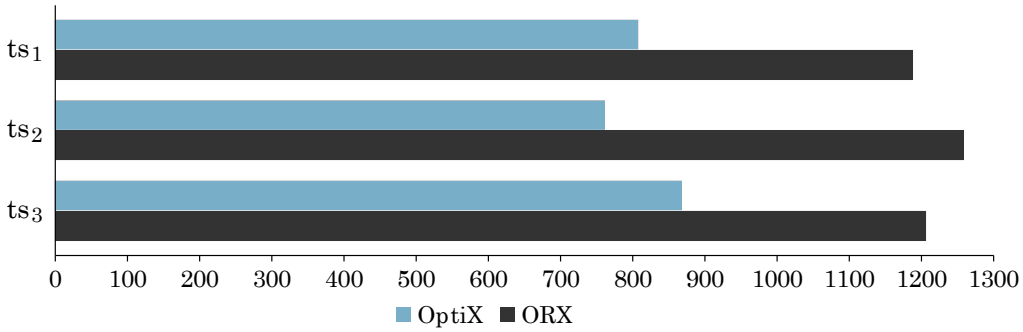


Figure 13: Results from Test Data Collection 1 in Experimental Evaluation

The complete results from the experimental evaluation are visualized in Figure 12. Because of major variations in numbers, a logarithmic scale is used in the graph. The results from each collection will subsequently be discussed individually.

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 an abundance of possible solutions.

As we can see from Figure 13, OptiX provided excellent results compared to ORX for these test sets. Not only were all of the execution times provided by OptiX between 305 and 472 seconds faster than ORX; it also provided better searching times. ts_1 was the test set with the most possible solutions, as all tests could be executed on all machines, and thus the only test set with which OptiX timed out after 30 seconds. 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 post-processing time being extended as a consequence of large numbers of tests and machines. This was the collection in which the results provided by OptiX was the most prominent.

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 test the mechanisms in situations with a large number of tests and a small number of machines.

Figure 14 shows that even though OptiX provided better total time than ORX on all test sets, the difference between the two mechanisms is less prominent in this collection than in the previous one. The largest difference in total time is 449.07 seconds for ts_6, and the smallest 102.11 seconds for ts_5, which are both small numbers considering that the total duration is

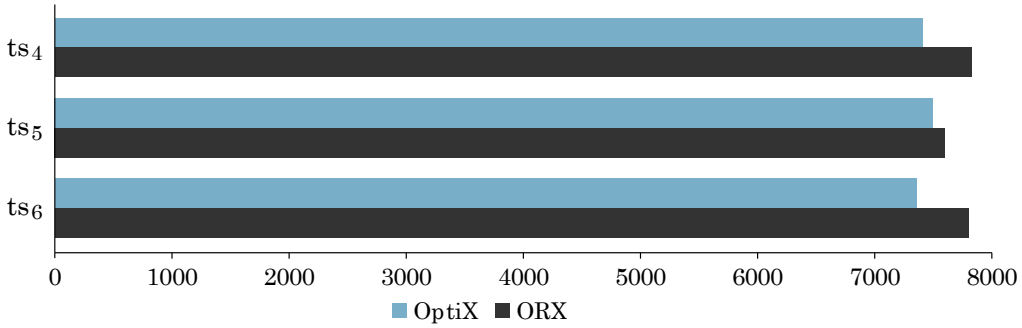


Figure 14: Results from Test Data Collection 2 in Experimental Evaluation

more than two hours for both mechanisms on each test set. The is likely linked to the much smaller number of possible solutions in the test sets in this collection compared to the first one. Nevertheless, OptiX undeniably outperformed ORX in searching time, where OptiX used less than a second on each test set and ORX timed out after 30 seconds, as well as in the execution time, and thus the total time on all test sets in this collection.

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 realistically number of tests, and a fair number of machines.

Figure 15 shows that OptiX once again provided better total results for all of the test sets in the collection. OptiX finished searching after 1.72, 0.22 and 0.67 seconds respectively for the three test sets, while ORX again was timed out after 30 seconds.

ts_7, which was the test set with the largest number of possible solutions,

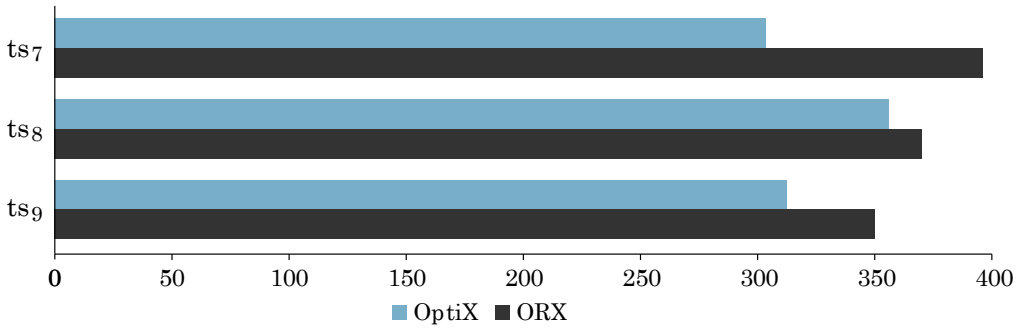


Figure 15: Results from Test Data Collection 3 in Experimental Evaluation

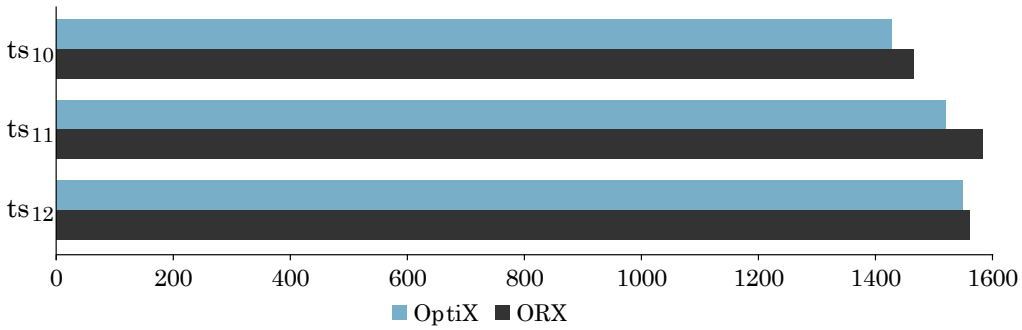


Figure 16: Results from Test Data Collection 4 in Experimental Evaluation

was also the one where the difference between the two mechanisms were the most distinct as the total difference was 92.67 seconds. The two remaining differences were not as conspicuous. An interesting trait that was discovered was that ORX provided a 17 second shorter execution time for ts_8 than OptiX accomplished. However, by using 30 seconds to find this solution, OptiX still provided a better total time. This demonstrates that it is not enough to find an allocation in which the execution time is minimized; it is also important to minimize the time taken to find the solution.

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

Again, OptiX produced better results for all of the test sets, which can be seen in Figure 16. As with collection 3, the differences in results were not as outstanding for this collection as for the two first ones. This is likely due to a much more limited amount of possible solutions to the problem, as the number of test machines was small and the number of tests moderate.

The largest difference in execution time was a mere 33 seconds for ts_{11}. Again, ORX produced a better solution of execution time for ts_{12}, but since OptiX used less than a second to find its solution for this and the remaining test sets, and ORX again was timed out after 30 seconds for the whole collection, OptiX provided the overall best result of 1549.80 seconds, which was 11.26 seconds less than the total time found by ORX for this problem.

OptiX provided better results than ORX on all of the test sets in the collection. The searching time was generally exceptional, aside from for

ts_1, where the searching was timed out after 30 seconds, and ts_2, where the searching time was more than 8 seconds. Compared to the searching times of ORX, OptiX did a better job on every test set.

It is a clear trend that ORX requires a lot of time to find good results, and that the quality of the final results most of the time is poor compared those of OptiX. Both mechanisms were timed out upon solving the problem with the largest amount of combinatorial solutions, but the total time of ORX' result was nevertheless 147% of that of OptiX'.

However, ORX found better execution times for ts_8 and ts_12, which means that even though OptiX provided better overall results, the mechanism still has potential for improvement. Although OptiX performed better that the benchmark values for all test sets, the difference between the results provided by the two mechanisms was generally not tremendous. This was especially the case for collection 4, which is the most realistic scenario covered by the experimental evaluation.

OR-tools is an excellent library for solving CP and combinatorial problems, but OptiX provided better results in this specific problem. A contributing factor to can be explained by OptiX being designed and implemented specifically to solve this type of problem efficiently, whereas OR-tools was designed to provide feasible solutions to a much broader range of problems. It is therefore not guaranteed to make the best decisions at any point of the process, so the time taken 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.

6.2 Threats to Validity

Judging from the results obtained in the previous section, it is clear that OptiX is performs superior 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 their 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 optimal. 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 their 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.

6.3 Discussion

TODO

6.4 Return on Investment

Altibox has great interest in how incorporating OptiRun to their testing process can affect their company from a business perspective. This section aims to enlighten 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.

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 be require excessive amounts of time and effort. Also, a brief user manual is included both on the website and in the appendix 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 library 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 that 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. Attempting to automate everything is also an approach that is almost guaranteed to fail [**?**].

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 with only a few keystrokes. The remaining time can thus be used for manually conducting the tests unsuited 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.

Except Safari?

Detecting more defects will likely result in improved product quality, and by extension additional time to test the product to an even greater extent. Other effects of this could be fewer customer inquiries and higher

customer satisfactory, 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.

7 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.

7.1 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.

7.2 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 [18]. It supports iOS, Android and FirefoxOS, and is compatible with Selenium Grid [19]. 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.

7.3 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) [20], or its successor, *Firebase Cloud Messaging* (FCM), [21], 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.

7.4 Continuous Integration

Test automation is commonly applied to continuous integration (CI) [6] 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 commonly used CI tool, offers a Selenium plugin [22] 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.

8 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.

TODO - Conclusion not finished

Appendices

A
Setup Instruction

TODO

B
User Manual

TODO

C
Test Script Template

This template for automated test scripts should be strictly conformed to:

```
1  import json
2  import sys
3  import unittest
4  from selenium import webdriver
5
6  class NameOfTest(unittest.TestCase):
7     @classmethod
8     def setUpClass(cls):
9        data = json.loads(sys.argv[1])
10
11       cls.driver = webdriver.Remote(
12          command_executor=data['command_executor'],
13          desired_capabilities=data['desired_capabilities']
14       )
15
16    def name_of_test(self):
17       """ Write the test script here """
18
19    @classmethod
20    def tearDownClass(cls):
21       if cls.driver is not None:
22          cls.driver.quit()
23
24 if __name__ == "__main__":
25    unittest.main(argv=['TestCase'])
```

Listing 11: Template for Automated Test Scripts

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.

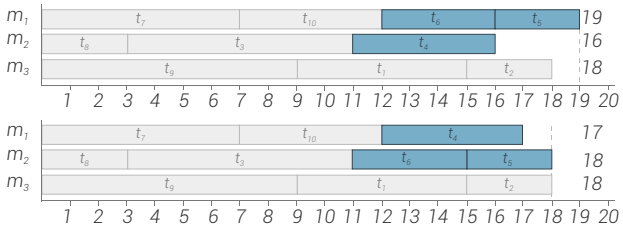
This thesis presents *OptiRun*; a platform where Altibox can run parallel tests in a distributed system. OptiRun is operated from a web-based interface that was developed as part of the thesis. A major objective has been to design and implement *OptiX*; a mechanism for allocating tests to machines in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

METHODOLOGY

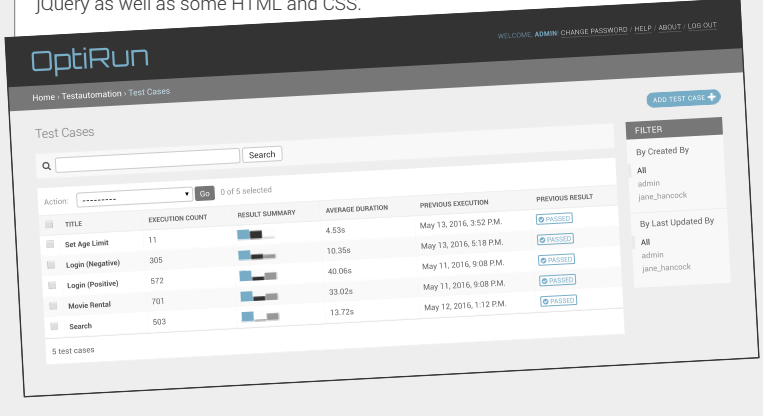
OptiRun consists of two main elements; a controller, which takes care of test allocation, execution and result reporting, and a web application where users can upload and manage test scripts, request test executions, view execution results and report failed test executions to the issue tracking system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



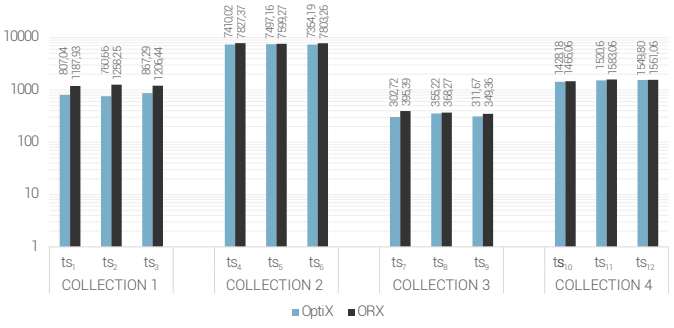
RESULTS

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 suites. The test suites in each collection all represent scenarios with a given number of tests and available machines. What separates the test suites in the same collection is the number of machines each test in the test suite 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 suites 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 suites.



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 suites. 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 suites.

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.

IPTV Department
Altibox AS

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

Figures

Listings

References

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

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

[3] Craig, R. D. and Jaskiel, S. P., 2002, *Systematic Software Testing*. URL: http://flylib.com/books/en/2.174.1.1/1/. [ONLINE], Accessed: 2016-03-14.

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

[5] 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*.

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

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

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

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

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

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

[12] *Selenium WebDriver — Selenium Documentation*. URL: http://www.seleniumhq.org/docs/03_webdriver.jsp, . [ONLINE], Accessed: 2016-02-02.

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

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

[15] 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.

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

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

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

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

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

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

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