TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Triin Vask IAPB134296

MODEL-BASED LOAD TESTING OF WEB APPLICATIONS: MOODLE WEB APPLICATION CASE STUDY

Bachelor's Thesis

Supervisor: Juhan-Peep Ernits
PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Triin Vask IAPB134296

MUDELIPÕHINE VEEBIRAKENDUSTE KOORMUSTESTIMINE MOODLE VEEBIRAKENDUSE NÄITEL

Bakalaureusetöö

Juhendaja: Juhan-Peep Ernits
PhD

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Triin Vask

08.01.2019

Abstract

Testing is an integral part of the software development life cycle. Performance evaluation and load testing are an important part in defining the quality of the software. To achieve that, traditionally scripted or pre-recorded tests are used to simulate the load on system.

This thesis suggests using NModel model-based testing framework for the purposes of load testing Moodle web application. Moodle is a learning-management system written in PHP. The goal of this thesis is to see whether model-based load testing can be used to find the underlying performance issues in Moodle.

In the first part of this thesis an overview about model-based testing and previous work done in the field of model-based load testing is given. The second part of the thesis introduces Moodle software and gives textual specification of the requirements of the system under test. The third part covers modelling the system under test using NModel framework followed by details on test setup in the next section. The final part covers the analysis of the test results and an assessment to whether model-based load testing is a valid choice.

This thesis is written in English and is 51 pages long, including 5 chapters, 31 figures and 4 tables.

Annotatsioon

Mudelipõhine veebirakenduste koormustestimine Moodle veebirakenduse näitel

Testimisel on tähtis osa tarkvara elutsüklis. Jõudlus- ja koormustestimine omavad tähtsat rolli tarkvara kvaliteedi määramisel. Traditsiooniselt kasutatakse selle saavutamiseks eelnevalt lindistatud teste või skripttestimist.

Antud bakalaureusetöös proovitakse jõuda selgusele, kas raamvara NModel saaks kasutada mudelipõhise veebirakenduste koormustestimise tarbeks kasutades Moodle õppekeskkonna veebirakendust testitava rakendusena. Töö eesmärk on näha kas mudelipõhiste koormustestidega on võimalik leida üles antud süsteemi kitsaskoht jõudluses.

Töö esimeses osas antakse ülevaade mudelipõhisest testimisest ja ka varasemastest töödest, mis on tehtud mudelipõhise koormustestimise valdkonnas. Teine peatükk kirjeldab Moodle süsteemi ja testitavale rakenduse osale kehtivaid nõudeid. Töö kolmas peatükk katab süsteemi modelleerimise kasutades raamistikku NModel. Töö neljas osa kirjeldab testkeskkonna ja testide ülesseadmist. Viimases peatükis antakse ülevaade testide tulemustest ja hinnang mudelipõhisele koormustestimisele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 51 leheküljel, 5 peatükki, 31 joonist, 4 tabelit.

List of abbreviations and terms

API	Application programming interface
BDD	Behaviour Driven Development
BDT	Behaviour Driven Testing
CLI	Command-line Interface
ct	Conformance Tester
dll	Dynamic Link Library
FSM	Finite state machine
GUI	Graphical User Interface
MBT	Model-based testing
mpv	Model Program Viewer
otg	Offline Test Generator
PHP	PHP: Hypertext Preprocessor
RUM	Realistic usage model
SDLC	Software development life cycle
SSH	Secure Shell
SUT	System under test
UI	User interface
URL	Uniform Resource Locator

Table of contents

Introduction 12
1 Model-based Testing 14
1.1 Overview 14
1.2 Benefits of model-based testing 15
1.3 Implementing MBT 17
1.4 Model-based load testing 18
1.5 NModel framework 19
2 Moodle Software 24
2.1 General overview 24
2.2 Load testing Moodle software 24
2.3 User roles in Moodle 25
2.4 Textual specification of the requirements of the SUT 26
3 SUT modelling 27
3.1 Functionality under test 27
3.2 Creating the model 28
3.3 Stepper 31
3.4 Adapter 32
4 Test setup 35
4.1 Server-side setup 35
4.2 NModel Metrics 35
4.3 Generating load using model-based tests 36
4.4 PHP Profiling 37
4.5 Atop monitoring 38
5 Test results 40
5.1 NModel metrics 40
5.2 Atop monitoring 41
5.3 PHP Profiling 45
5.4 Assessment of model-based load testing 46
Summary 47

References 48
Appendix 1 – Source code to the Moodle model and adapter.. 50
Appendix 2 – Server parameters 51

List of figures

Figure 1. General model-based testing setting [5] 15
Figure 2. V-diagram showing traditional software project activities and schedule. [3]. 16
Figure 3. V-diagram showing opportunities for model-based testing and analysis. [3]. 16
Figure 4. Work procedure in LTAF [8] 18
Figure 5. A sample of the sequential action model [9] 19
Figure 6. An FSM generated by *mpv* [3] 21
Figure 7. Running the *otg* tool 22
Figure 8. Output file generated by the *otg tool* 22
Figure 9. Output file generated by the *ct* 23
Figure 10. Moodle login code in the SUT model 29
Figure 11. FSM illustrating login 30
Figure 12. State parameters in login action 30
Figure 13. Full FSM illustrating the SUT 31
Figure 14. Stepper interface 31
Figure 15. Additional methods in adapter 33
Figure 16. Example of a compound term 34
Figure 17. Login action in adapter 34
Figure 18. Test setup 35
Figure 19. NModel metrics output 36
Figure 20. Model size with 4 users 36
Figure 21. Bash script to start multiple instances of the *ct* 37
Figure 22. Screenshot of webgrind UI 38
Figure 23. Bash script for graphing Atop log files 39
Figure 24. CPU load average for 1 user 41
Figure 25. Disk busy and MySQL CPU usage percentages for 1 user 42
Figure 26. CPU load average for 30 users 42
Figure 27. Disk busy and MySQL CPU usage percentages for 30 users 43
Figure 28. CPU load average for 40 users 44
Figure 29. Disk busy and MySQL CPU usage percentages for 40 users 44

Figure 30. Profiling results for 1 user 46
Figure 31. Profiling results for 30 users 46

List of tables

Table 1. Steps in implementing MBT [7] 17
Table 2. User-roles in Moodle [16] 25
Table 3. Actions, corresponding actions in model and state variables 27
Table 4. NModel metrics time spent in each action in seconds. 40

Introduction

Software testing is an integral part of the software development life cycle (SDLC). Testing can help verify that the software conforms to the requirements and help validate that the requirements for intended use of the application have been fulfilled.

SDLC consists of gathering the requirements, development according to those requirements, testing the new features, delivery of the developed and tested features, and feedback. Testing can be applied throughout the SDLC. Requirements can be tested using requirement analysis to see that all possible scenarios have been considered as well as to check the consistency of the requirements and to find contradictions [1]. Unit testing and static analysis can be used to test the code. Manual and automated testing is used to verify that the application is working properly. The previously mentioned test techniques belong under functional testing techniques. There is another group of testing called non-functional testing. Non-functional testing checks non-functional aspects such as performance, usability, load, reliability, security etc.

In this thesis, we will be looking at model-based load testing to see whether model-based testing (MBT) could be used for load testing purposes. MBT is a type of functional testing that has been gaining popularity over the years. The possible benefits of using MBT are better test coverage, easier maintenance and increased fault detection. Additionally, using MBT for the purposes of load testing could help create more lifelike load on the system.

The system under test (SUT) is Moodle web application with known performance issues. The goal of this thesis is to see whether model-based load testing using NModel framework could be used to detect the performance issues affecting Moodle web application. To achieve that, model-based tests will be generated using NModel and then multiple instances of those tests run simultaneously. The performance metrics will be collected on the server running the system under test by using Atop and PHP profiling.

This thesis is divided into 5 chapters. In the first chapter there will be a more thorough overview about model-based testing, benefits of MBT, general guidelines on

implementing model-based tests, overview about other work done in the field of model-based load testing and intro to NModel framework. The second chapter covers Moodle software – overview, load-testing done on Moodle, user roles in Moodle and textual specifications of the functionality under test. The third chapter dives into modelling the system and creating the model-based tests. The fourth chapter covers test setup such as the server-side setup and the setup of the environment and tools to execute the tests. The final chapter covers running the tests and analysis of the test results as well as an assessment whether using NModel for the purposes of load testing is a valid choice.

1 Model-based Testing

The following chapter gives a general overview about model-based testing (MBT), the benefits of MBT, outlines for implementing model-based tests, a brief overview about previous work done in the field of model-based load testing and an intro to the model-based testing framework NModel.

1.1 Overview

Model-based testing refers to the process and techniques for the automatic derivation of test cases from models, the generation of executable scripts, and the manual or automated execution of the resulting test cases or test scripts [2].

A model describes the intended behaviour of the system under test (SUT). A model is a simplified version of the SUT and it does not need to describe the whole system but can be used to describe a program unit or a component. It is impossible to consider every implementation detail and therefore during model generation, it is generally decided what details to omit and what to simplify. *Abstraction* of the model program defines what details to include. The higher the level of abstraction, the more details have been omitted from the model and the simpler the model is.

Abstractions can be categorized into three types. *Data abstraction* deals with variables. The higher the abstraction level, the fewer variables and values are used. *Behavioural abstraction* deals with statements and methods. Higher behavioural abstraction means that each statement and method covers more functionality in the SUT. The last type of abstraction is *environmental abstraction* which deals with control structure leaving it to the tool or environment to decide which methods to execute [3].

There are two main ways of executing model-based tests – online and offline [4]. In *offline* or *a priori* testing, test cases are generated before execution. Offline testing can be further divided into offline generation of executable tests and offline generation of manually deployable tests. The former meaning that the MBT tool generates tests in

computer-readable form that can be later run automatically and the latter meaning that the test sequences are generated in human readable form to be executed manually. *Online testing* or *on-the-fly testing* means that the test cases are generated as the test runs. In both techniques, test cases are generated by exploring the model program [3].

To execute model-based tests a test harness is required. A test harness is code that enables a test runner to execute actions in the system under test. It works as a bridge between the model and the system itself.

Figure 1 describes the general principles behind model-based testing.

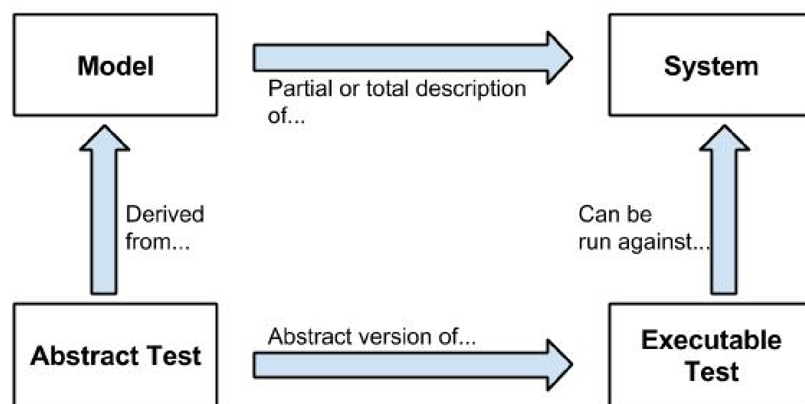

Figure 1. General model-based testing setting [5]

1.2 Benefits of model-based testing

There are multiple benefits of using model-based testing instead of more traditional approaches. One of the main benefits is that checking, and testing development products can happen earlier in the SDLC. Figure 2 illustrates traditional software project activities and schedule.

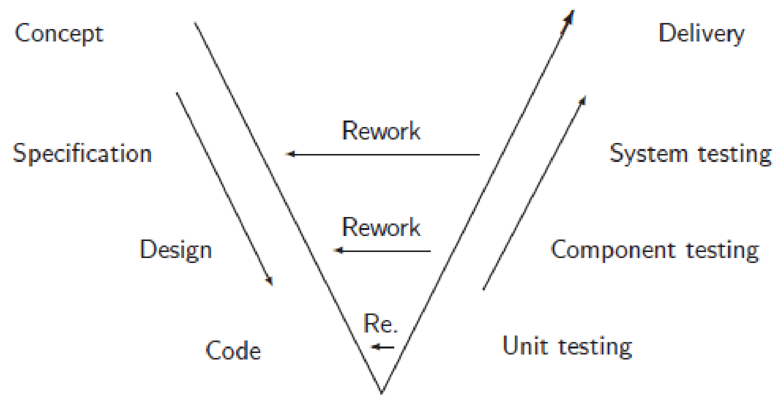

Figure 2. V-diagram showing traditional software project activities and schedule. [3]

The traditional V-diagram (Figure 2) shows how each testing activity on the right side corresponds to development activity at the same level on the left side. The issue with this kind of approach is that problems arising from first products on the left might not be discovered until much later in SDLC making them exponentially costlier to fix.

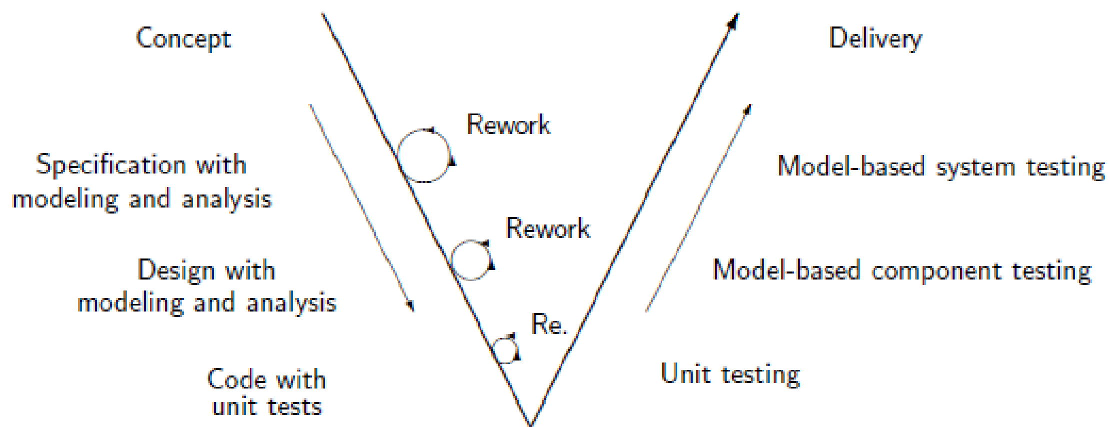

Figure 3. V-diagram showing opportunities for model-based testing and analysis. [3]

Model-based testing and analysis enables us to gain immediate feedback earlier in the development. Analysis with model programs can be used to check specifications and designs the same way unit tests check code hence making it possible to fix problems immediately [3]. Figure 3 illustrates the modified V-diagram.

Other possible benefits of using model-based testing are [4] [6]:

- Easier test case/suite maintenance – changes to the SUT can be implemented with less effort.

- Cost reduction
- Improved test coverage
- Improved testing efficiency and quality
- Increased fault detection – random execution of the model can reveal problems that otherwise could go unnoticed
- Establishing templates or specifications around intended behaviour of the product

1.3 Implementing MBT

The process of implementing MBT can be divided in multiple ways. The following table (Table 1) describes one possible division. On the left side is the step followed by the actions to be taken in that step on the right side.

Step	**Actions**
Understanding the system	Deciding what aspect of the system to model. Reading specifications. Exploring the system/exploratory testing. Optionally dividing system into multiple parts.
Choosing modelling framework	Deciding what requirements the framework must conform to (e.g. non-determinism, paradigm, test selection criteria, test generation technology, language it is written in, etc.)
Creating model	Identifying system inputs. Defining input's allowed values, boundary values, invalid values and expected responses. Defining state transitions.
Generating test cases	Deciding on coverage level (e.g. *all states coverage* or *all transitions coverage*). Automatically generating the test cases based on coverage rules.
Test execution	Linking the model to some kind of testing interface (e.g. API or GUI test automation interface).

Table 1. Steps in implementing MBT [7]

1.4 Model-based load testing

Using model-based testing for load testing is a new concept that has not been used much. In 2010, a paper "Model Based Load Testing of Web Applications" was written by Xingen Wang, Bo Zhou, Wei Li [8]. The paper proposes using usage models and model-based tests as an alternative to traditional load testing as it enables generating load tests where virtual user's behaviour is closer to that of an actual user's. For the purposes of modelling, realistic usage models (RUM) based on unified modelling language (UML) are suggested. The key concept of using RUM relies on taking user scenarios describing a single user in a simple way, mapping them to an Activity Diagram and then mapping action scripts to those models. To produce realistic and accurate loads, a simple load model (SLM) was used together with RUM. In SLM, there are two parameters used: the proportion of each action in RUM and the think time of the actions in RUM. The SLM and RUM were implemented into load testing tool Load Testing Automation Framework (LTAF).

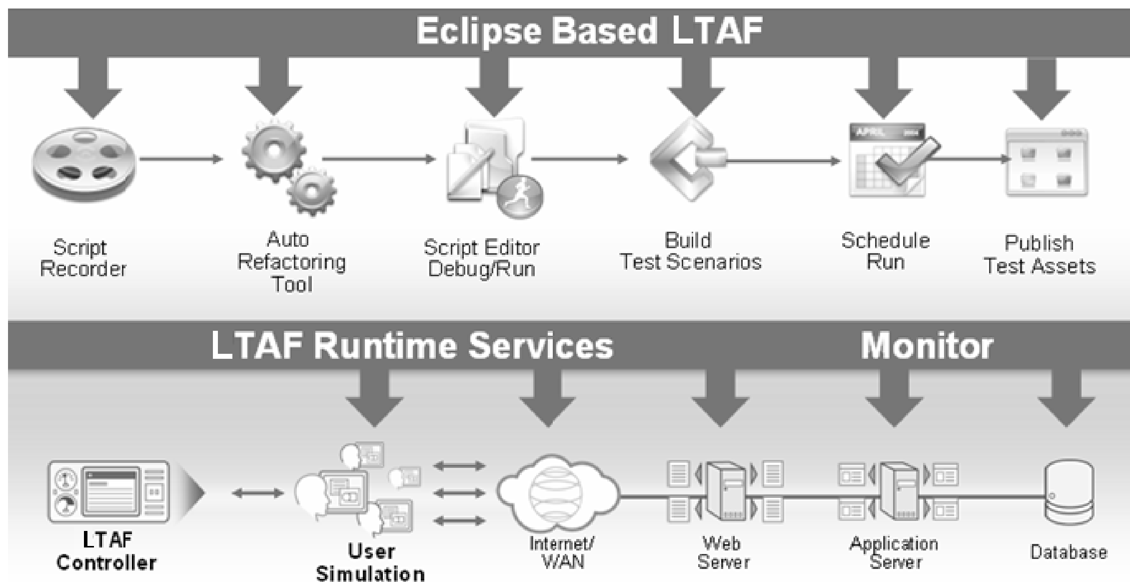

Figure 4. Work procedure in LTAF [8]

LTAF is described in the paper as: "*LTAF starts performance test by recording the single visitor's behaviors into test scripts, then refactor the recorded scripts and do some parameter works on the recorded scripts, and then create RUM visually, fill the proportion of general sessions with SLM if server logs exist, and then create runtime settings to configure related system stress, and at last run the scripts to simulate multiple users. You can also see the work procedure in Figure 8 (see* Figure *4). The relevant test*

result can be analysed with the analysing tools, which is also provided in LTAF." [8] It was also noted that the load times for the model could be the area for future improvement.

In 2014, a conference paper "LTF: A Model-based Load Testing Framework for Web Applications" by Junzan Zhou et al. was published [9]. The paper looks into modelling system workload and generating synthetic web workloads. The authors point out that traditional generative models are not generic enough to be applied for load testing and suggest using a Context-based Sequential Action Model (CBSAM) to describe users application usage patterns. They suggest using Workload Parameter Specification Language (WPSL) that works as a link between the CBSAM and the system under test. Figure 5 is an example of a simple sequential action model. Each action has a set probability derived from the SUT logs – additional information can be added to specify context. Additionally, a Load-Testing Framework was introduced to implement CBSAM and WPSL and it was concluded that the framework created could be used to generate accurate, stable and reliable synthetic workloads [9].

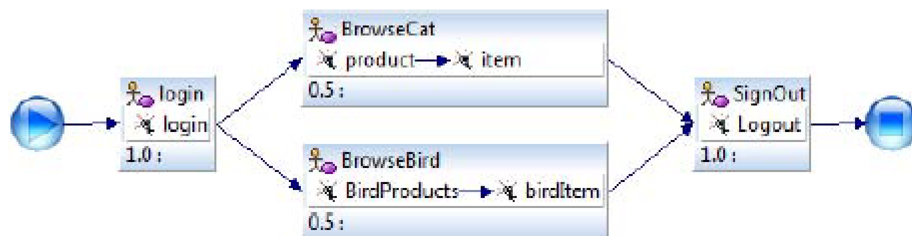

Figure 5. A sample of the sequential action model [9]

1.5 NModel framework

There is a large number of MBT tools developed to support the practice of MBT. In this thesis NModel will be used, specifically a branch of NModel which supports writing models in programming languages C# and F# [10]. The motivation behind choosing NModel is mostly practical. Firstly, NModel conforms to all needs – it has the option of on-the-fly testing, it is open-source and it has tools to visualize and validate the model. Additionally, there is previous work done combining NModel and Moodle Mobile Application by Gabriel Kolawole in master's thesis "Model based testing mobile applications: A case study of Moodle mobile application" [11] making it a good reference point when it comes to generating the model of the SUT enabling to shift focus from modelling to load generation using models.

NModel is an open-source framework that was designed and implemented at Microsoft Research by Colin Campbell and Margus Veanes. It is thoroughly explained in the book "Model-based Software Testing and Analysis with C#" by Jonathan Jacky, Margus Veanes et al. [3] and the book is used as a reference in this thesis.

Some of the highlights of NModel are [12]:

- Open-source software;
- Modelling using abstract data structures (sets, maps, bags) and objects;
- Composition of model programs. Models can be split up into features that can be composed into a single model;
- Exploration of model programs using Model Program Viewer;
- Model programs can be written in C# and F#;
- Supports non-determinism and asynchronous testing.

The NModel framework comes with many tools to ease the MBT process: a visualization tool *mpv* (Model Program Viewer), test generation tool *otg* (Offline Test Generator), and test runner tool *ct* (Conformance Tester).

Design errors in model program can be found by using *mpv* tool to explore, search and display the finite state machine (FSM). An FSM displays all the possible runs of the system. The following figure (Figure 6) illustrates an FSM generated by the *mpv* tool where safety analysis has detected unsafe states and filled them in. Additionally, the tool provides a written report upon finding such states.

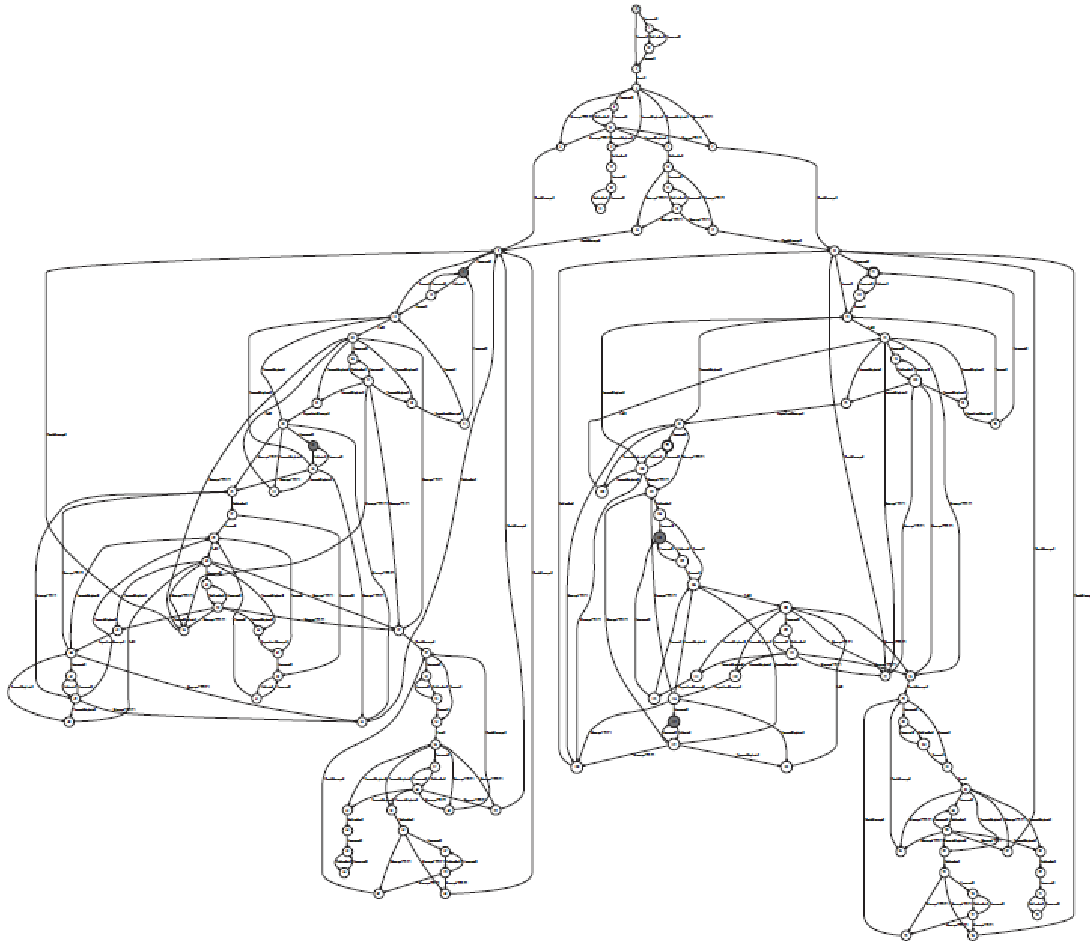

Figure 6. An FSM generated by *mpv* [3]

The *mpv* tool can also search for dead states from which none of the accepting states can be reached such as deadlocks (a loop with no way forward) and livelocks (where program cycles without making any progress). By default, the FSM generated by *mpv* expresses all possible behaviours of the modelled system [3] up to a maximum limit of states and transitions selected by the user in the UI (note that the labelled transition systems generated from model programs can be infinite).

The *otg* tool explores the model program, creates an FSM and then traverses through the FSM and saves the paths to a file. The *otg* traverses through the FSM using postman tour eliminating all paths to dead states trying to cover all possible transitions in the system. To generate the test suites, otg.exe must be run on command line with arguments specifying the model program dll (dynamic-link library) and output file. Figure 7 illustrates one possible way of running the tool.

```
otg /r:MoodleModel\\bin\\Debug\\MoodleModel.dll MoodleModel.Factory.Create
    /file:testsuite.txt
```

Figure 7. Running the *otg* tool

The *otg* saves the test suites to the specified file in the following format illustrated by Figure 8:

```
TestSuite(
    TestCase(
        login_start(User("Student"), Password("Incorrect")),
        login_finish(User("Student"), LoginStatus("Failure")),
        login_start(User("Student"), Password("Correct")),
        login_finish(User("Student"), LoginStatus("Success")),
        search_start(SearchKey("ValidCourse")),
        search_finish(SearchKey("ValidCourse"), SearchStatus("Found")),
        enrol_start(User("Student"), EnrolmentKey("Incorrect")),
        enrol_finish(User("Student"), EnrolmentStatus("Failed")),
        enrol_start(User("Student"), EnrolmentKey("Incorrect")),
        enrol_finish(User("Student"), EnrolmentStatus("Failed")),
        logout_start(User("Student"))
    )
)
```

Figure 8. Output file generated by the *otg tool*

The test runner tool *ct* can be executed with both output file from the *otg* and on the fly where test cases are generated as the *ct* runs. Running the *ct* is similar to running the *otg* tool – it takes model, stepper and test file as arguments. If log file is given then test results are logged into a file, otherwise they are logged in the console window. Figure 9 illustrates a typical output generated by the *ct* showing both successful and failing test results.

```
TestResult(17, Verdict("Failure"), "no such element: Unable to locate
element: {\"method\":\"xpath\",\"selector\":\"//li[@id='module-
5']//div//div/a\"}
    Trace(
        login_start(User("Student"), Password("Correct")),
        login_finish(User("Student"), LoginStatus("Success")),
        search_start(SearchKey("ValidCourse")),
        search_finish(SearchKey("ValidCourse"), SearchStatus("Found")),
        enrol_start(User("Student"), EnrolmentKey("Correct")),
        enrol_finish(User("Student"), EnrolmentStatus("Successful")),
        quiz_start(User("Student"))
    )
TestResult(18, Verdict("Success"), "",
    Trace(
        login_start(User("Student"), Password("Incorrect")),
        login_finish(User("Student"), LoginStatus("Failure")),
        login_start(User("Student"), Password("Correct")),
        login_finish(User("Student"), LoginStatus("Success")),
        logout_start(User("Student")),
        logout_finish(User("Student"))
    )
```

Figure 9. Output file generated by the *ct*

Modelling using NModel will be covered in a later chapter under SUT modelling.

2 Moodle Software

The following chapter gives a brief overview of the Moodle application describing the functionality, user roles and functional requirements necessary for this thesis.

2.1 General overview

Moodle is a free and open-source learning management system designed to provide educators, administrators and learners with a system to create personalised learning environments [13]. Moodle is highly customizable and can be tailored to individual needs. Moodle supports OAuth 2 services, enabling users to log in using different services such as Google, Microsoft and Facebook as well as different plugins and features.

In SUT there are two options for logging in - using email/username and password, and logging in using Office 365 accounts. To reduce the complexity of the SUT, we will only look at username/email authentication to avoid possible OAuth2 related issues. Using only username/email authentication is sufficient because the main purpose of this thesis is to concentrate on Moodle-related functionality.

2.2 Load testing Moodle software

Load testing helps to identify the maximum operating capacity of an application and find bottlenecks. Usually, the motivation behind load testing Moodle is to determine whether the system will perform under the intended load of users performing specific tasks. Load tests can also be used when performance issues have already arisen to help find the root cause which is the case with the SUT.

JMeter is the most popular choice of software used to load test Moodle. It can be used to simulate a heavy load on server, group of servers, network or object [14]. Test cases are usually scripted to simulate users performing different actions in the system. Once the scripts are working, the tests are run with a set of different users and system performance under load is monitored. Load testing Moodle is explained more thoroughly in a blog post

“Load-testing Moodle 2.6.2 at the OU” written in 2014 by a Moodle developer, Tim Hunt [15].

There has been no documented model-based load testing performed on Moodle software. The objectives of model-based load testing are similar to regular load testing – generating load and observing what happens to the performance of the SUT. The possible benefit of using models with load testing is the hope of achieving more life-like load simulation.

2.3 User roles in Moodle

There are several user roles in Moodle with different permissions. Table 2 gives a brief overview of the roles and their description.

There are following user roles in Moodle:

Role	**Description**
Site administrator	Superuser. Has permissions to do everything.
Manager	Lesser administrator role.
Course creator	Can create courses.
Teacher	Can manage and add content to courses.
Non-editing teacher	Can do grading in courses but not edit them.
Student	Can see available courses, participate in course activities and view resources but cannot alter them or see the class gradebook. Can see their own grades if teacher has allowed it.
Guest	Can view courses but cannot participate.

Table 2. User-roles in Moodle [16]

This thesis will only concentrate on the roles of student and site administrator. The requirements of the student role will be covered in the next section. Site administrator role will be used to reset accessed courses to the initial state by unenrolling students and will not be covered more thoroughly.

2.4 Textual specification of the requirements of the SUT

Using textual requirement specifications relates strongly to behaviour driven development (BDD) and testing (BDT). BDT is focused on the behaviour of the users rather than the technical functions of the software. Requirements in human-readable form can be used to validate the system's functionality [17]. The implementation of BDT follows roughly the following steps – formal user stories are written from which test scenarios are derived by the tester, the scenarios are reviewed and then implemented by the tester. In this thesis, textual specification of the student user role is used as a reference point when creating the SUT model.

User role: *Student*

1. User chooses Moodle account as authentication option. Username and password input fields are displayed.
2. User enters username and password and clicks Log in button.
3. If login is successful then grades, calendar, notifications, profile information and enrolled courses are displayed to the user. If login is unsuccessful, the user is shown the login page again.
4. User enters keyword to the course search box.
5. If the keyword is valid then matching courses are displayed, otherwise no results are displayed.
6. User clicks on course.
7. If enrolment is required, then then user must enter an enrolment key.
8. If the enrolment key is correct, then the user can access course materials. If the key is incorrect then the user is shown the enrolment page again.
9. User clicks on quiz link.
10. Quiz attempts page is displayed.
11. User clicks on quiz start button.
12. Quiz is displayed.
13. User fills in answers to each question using radio buttons.
14. User submits the answers and confirms his submission.
15. Test results are displayed.
16. User clicks the logout button.
17. If logout is successful, then user cannot see authenticated user content anymore.

3 SUT modelling

The following chapter gives an overview about the actions taken to create model-based tests for the Moodle web application. It covers choosing the functionality to be tested, building the model in F# using NModel framework, validating the model using *mpv*, and building the system adapter and stepper using F#. The process is illustrated using the login functionality.

3.1 Functionality under test

The functionality tested in this thesis is covered in Chapter 2 under *Textual specification of the requirements of the SUT*. All the actions described in requirements are implemented in the SUT model. Table 3 gives an overview of actions, their corresponding actions in the model and state variables associated to those actions.

Action	**Corresponding action in model**	**State variables**
Login	login_start login_finish	view activeLoginRequest usersLoggedIn
Logout	logout_start logout_finish	view activeLogoutRequests usersLoggedIn
Search	search_start search_finish	view currentSearch foundCourse
Enrol	enrol_start enrol_finish	view activeEnrolRequests courseParticipants enrolStarted
Take quiz	quiz_start quiz_finish	view quizStarted

Table 3. Actions, corresponding actions in model and state variables

Each action in the system (login, logout, etc) is divided in two in the model. Such division makes it easier to track whether the action was executed successfully – first by checking the enabling conditions for start and later for finish.

3.2 Creating the model

The model was created by using an iterative approach taking the following steps: picking an action to implement, deciding on what state variables to use, their value at state, and figuring out the enabling condition. High abstraction level was chosen for the model.

There are some similarities in the SUT model to the model created by Gabriel Kolawole in his master's thesis [11] when it comes to login, search and enrol actions. The work of Gabriel was a good reference point as the general functionality of the SUT was derived from Moodle application in both cases. Actions not covered in Gabriel's work but implemented in this thesis include logout and quiz actions.

In NModel the actions in model code are labelled with an attribute tag [<Action>]. Each action must have an enabling condition in the form of a method that returns true/false. The enabling condition should have the same name as the action with the suffix *Enabled*. An enabling condition describes a set of states where the action is allowed. It is possible to code enabling conditions such as more than one action is enabled in some states meaning that any of the enabled actions can lead to the next state.

Both actions and enabling conditions can have parameters. The parameters of an enabling condition must correspond to the action parameters – meaning that an enabling condition cannot have parameters that are not used in action method, but it may have fewer parameters than its action method. [3]

Figure 10 shows the code related to Moodle login actions in the SUT model.

```
// state variables
// view - current web page displayed.
static member val view = View.Login with get, set
// activeLoginRequests - map of users who have started login and their
login status.
static member val activeLoginRequests = Map<User,LoginStatus>.EmptyMap with
get, set
// usersLoggedIn - a set of currently logged in users.
static member val usersLoggedIn : Set<User> = Set<User>.EmptySet with
get,set

[<Requirement(Id = "Login start", Summary="The starting action for login.
Takes two arguments: user (username) and password (correct or incorrect).
The action is enabled when the current view is login screen and the current
user is not logged in.")>]
[<Action>]
static member login_start (user : User, password : Password) =
      Contract.view <- View.DashboardAuthenicated
      if password = Password.Correct then
            Contract.activeLoginRequests <-
            Contract.activeLoginRequests.Add (user,LoginStatus.Success)
      else
            Contract.activeLoginRequests <-
            Contract.activeLoginRequests.Add (user,LoginStatus.Failure)
static member login_startEnabled (user : User) =
      Contract.view = View.Login &&
      Contract.usersLoggedIn.Contains(user) = false

[<Requirement(Id = "Login finish", Summary="The finish action for login.
Takes two arguments: user (username) and loginStatus (success or failure).
The action is enabled when login request has been started for the user and
the view is authenticated user dashboard.")>]
[<Action>]
static member login_finish(user : User, loginStatus : LoginStatus) =
      Contract.activeLoginRequests <-
      Contract.activeLoginRequests.RemoveKey(user)
      if loginStatus = LoginStatus.Success then
            Contract.usersLoggedIn <- Contract.usersLoggedIn.Add(user)
      else
            Contract.view <- View.Login
static member login_finishEnabled(user : User, loginStatus : LoginStatus) =
      Contract.activeLoginRequests.Contains(Pair<User, LoginStatus> (user,
      loginStatus)) &&
      Contract.view = View.DashboardAuthenicated
```

Figure 10. Moodle login code in the SUT model

Login_start method takes two arguments – user and password. The password can be either correct or incorrect. The enabling condition for login_start is that the current view must be login page and the user must not be already logged in. When login is started with correct password then the expected outcome is successful login. The login_finish method checks if the action was finished successfully by the SUT taking the user and login status as arguments. The enabling condition for login_finish is that login request must be started for the user and the view is the logged in user dashboard.

When running the login model code in the *mpv,* following FSM was created (Figure 11).

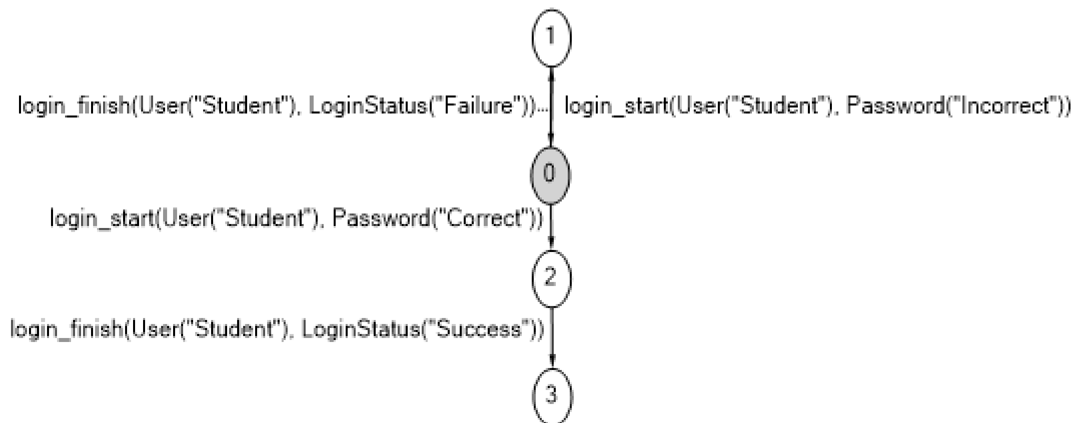

Figure 11. FSM illustrating login

Initial state of the SUT is illustrated with a grey circle. If login is started with the correct password, then the system goes from states 0 -> 2 -> 3 at the end of which the user is authenticated. If login is started with incorrect password, then the system goes through states 0 -> 1 -> 0 and the user ends up at the login page unauthenticated. The possible parameters in states are derived from enumerations defined in model program given in the following form (Figure 12):

```
type User = Student = 0
type Password = Correct = 0 | Incorrect = 1
```

Figure 12. State parameters in login action

Every action adds complexity to the model making the FSM more difficult to read and manage. Dividing the model to features helps reduce the complexity of the individual FSM. Features can later be combined when running the tests to achieve the desired system coverage. Figure 13 shows the full FSM of the Moodle application for one user.

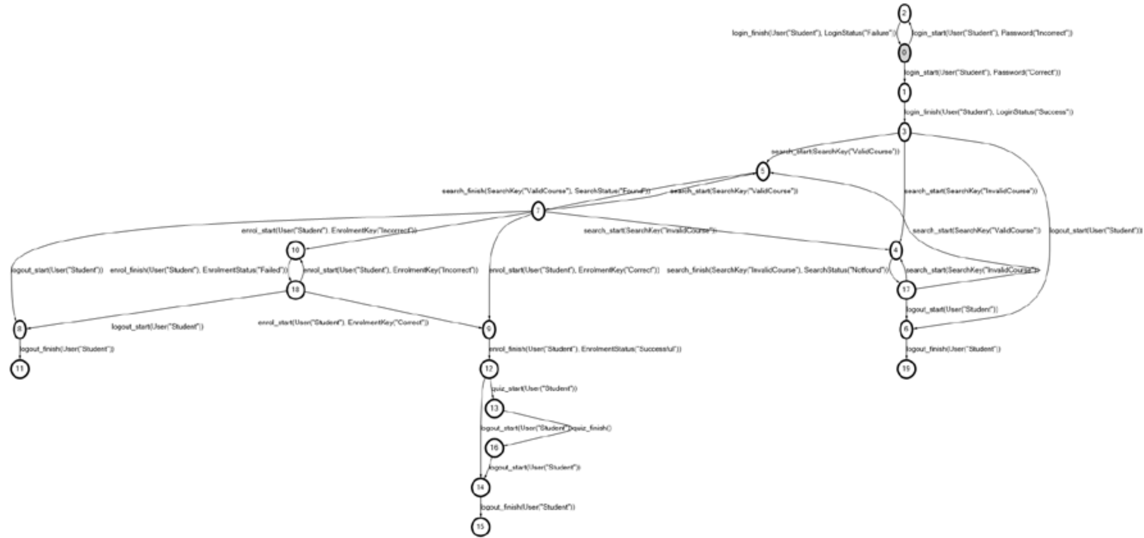

Figure 13. Full FSM illustrating the SUT

3.3 Stepper

Test harness called the stepper is the glue between the model and system under test. It connects the actions in the model to the real actions in the SUT. Figure 14 shows the stepper interface used.

```
interface IStepper with
      member this.DoAction (t : CompoundTerm) =
            match t.FunctionSymbol.ToString() with
            | "login_start" -> moodle.Login (t)
            | "logout_start" -> moodle.Logout (t)
            | "enrol_start" -> moodle.Enrol (t)
            | "search_start" -> moodle.Search (t)
            | "quiz_start" -> moodle.Quiz ()
            | s -> failwith (sprintf "Unknown action, %s" s)
// Reset does the following actions: 2s wait time, calls SavePageSource which
then saves page source to a file if the test failed, calls UnEnrol which
resets the test data to the initial state, discards previous driver instance
and initializes a new instance.
      member this.Reset() =
            System.Threading.Thread.Sleep(2000)
            moodle.SavePageSource ()
            moodle.UnEnrol ()
            Moodle.driver.Quit()
            moodle.Init ()
```

Figure 14. Stepper interface

When the model sends the action login_start to the stepper in form of a compound term then moodle.Login () is called in the SUT.

3.4 Adapter

The previous topic covered the stepper which is the harness between the SUT model and the system under test. The actions called in the stepper under DoAction method are implemented in the system adapter using Selenium UI testing framework. Selenium is an open-source software that can be used to automate activities in web browser and its primary application is automating web applications for testing purposes. There are two options for running Selenium – recording the actions in browser and using playback or scripting the tests. Selenium supports a number of programming languages making it easy to use with different languages and frameworks. The motivation for choosing Selenium was that it could be used with .NET and the author's previous experience using it. The adapter was written in F# and scripted testing was used to simulate user actions in browser.

The behaviour in the adapter is derived from textual specifications given under chapter 2.4 and contains some additional functionality such as the setup of the ChromeDriver (tool for automated testing of web applications which provides capabilities for navigating web pages, user input, JavaScript execution, etc.) in the Init method, SavePageSource method which logs page source as html when a test fails, SetXDebugCookie method to start PHP profiling using XDebug and UnEnrol method to reset the test data to the initial state by unenrolling the user after a test run has completed. Figure 15 illustrates the methods described previously with some additional comments. In the Init method there is a check to see if the current user is "Tudeng" as the XDebug cookie should be set only in case of a specific user to avoid profiling data overload. The SavePageSource method checks if the web browser URL (uniform resource locator) contains .php extension to see if the test run finished successfully: the final page after logout should not contain the extension hence the check helps to see if the test run was successful and if it was not, saves the page source to a file. SetXDebugCookie is a simple PHP page which adds a cookie to the browser session which works as a flag to start PHP profiling. Profiling PHP and XDebug are covered more thoroughly in the next chapter under "Test setup".

```
member this.Init () =
    // set the chrome version to latest
    options.AddAdditionalCapability(CapabilityType.Version, "latest", true)
    // set the operating system
    options.AddAdditionalCapability(CapabilityType.Platform, "WIN8", true)
    // run chrome driver without graphical ui and disable gpu
    options.AddArgument("headless")
    options.AddArgument("disable-gpu")
    // path to chrome driver
    Moodle.driver <- new ChromeDriver("MoodleTest" +
                  string System.IO.Path.DirectorySeparatorChar), options)
    //time to wait for elements to appear
    Moodle.driver.Manage().Timeouts().ImplicitWait
                        <- TimeSpan.FromSeconds(5.0)
    // check if argument given to adapter is "Tudeng" - if yes, then init
    XDebug
    if arg.IsSome then
        let argValue = Option.get arg
        if argValue = "Tudeng" then
            this.SetXDebugcookie ()
    // open moodle home page
    this.OpenMoodle()

member this.SavePageSource () =
    match Moodle.driver.Url.Contains(".php") with
        | true -> let pageSource = Moodle.driver.PageSource
                    File.WriteAllText ("Logs" +
                    string (System.IO.Path.DirectorySeparatorChar) +
                    "pagesource-" +
                    System.DateTime.UtcNow.ToString("yyyyMMddHHmmss") +
                    ".html", pageSource)
        | _ -> ()

member this.SetXDebugcookie () =
    Moodle.driver.Navigate().GoToUrl(COOKIE_SITE)
    System.Threading.Thread.Sleep(2000)

member this.OpenMoodle () =
    Moodle.driver.Navigate().GoToUrl(MOODLE_SITE)

member this.UnEnrol () =
    // see if user is currently still logged in and logout
    let elementVisibleWhenUserLoggedIn =
        Moodle.driver.FindElementsById(itemVisibleWhenLoginSuccess)
    if elementVisibleWhenUserLoggedIn.Count > 0 then
        this.LogoutHelper ()
    // Log in as admin
    this.LoginAsAdmin ()
    // Go to course admin page
    Moodle.driver.Navigate().GoToUrl(COURSE_ADMIN_SITE)
    let unenrolUser =
            Moodle.driver.FindElementsByXPath("//div[@data-fullname='" +
            param_fullname + "']/a[@data-action='unenrol']")
    // Unenrol current user
    if unenrolUser.Count > 0 then
        unenrolUser.Item(0).Click()
        System.Threading.Thread.Sleep(2000)
        let submit =
                Moodle.driver.FindElementByXPath("//div[@class='modal-
                content']/div[@class='modal-footer']/button[1]")
        submit.Click()
    ()
```

Figure 15. Additional methods in adapter

The Login method called from the Stepper takes one argument in the from of a compound term passed from model to stepper and then on to the adapter. The compound term consists of the corresponding action name in the model and the current arguments. In the following example of a compound term *login_start*, it is given that login should start with the default user and with a correct password. Figure 16 illustrates a compound term sent to the adapter.

```
login_start(User("Student"), Password("Correct"))
```

Figure 16. Example of a compound term

The Login action consists of the following steps: Finding the input fields for username, password and submit button in the web page; checking for the value of "Password" in the compound term to decide whether the attempt at login should happen with the correct or incorrect password; inserting input values to corresponding input fields; submitting them and then validating the result in browser. A new compound term, *login_finish*, is returned to the stepper and model to cross-validate if the action call was successful and matches the rules set in the model. Figure 17 illustrates the Login action in the adapter.

```
member this.Login (t:CompoundTerm) =
    // Find the input fields and login button in web page
    let usernameField = Moodle.driver.FindElementById(username)
    let passwordField = Moodle.driver.FindElementById(password)
    let loginButton = Moodle.driver.FindElementById(loginButton)
    // Insert username
    usernameField.SendKeys(param_moodle_username)
    // Check whether the password to insert should be correct or incorrect
    if (string((t).[1]) = "Password(\"Correct\")") then
        passwordField.SendKeys(MOODLE_PASSWORD_CORRECT)
    else
        passwordField.SendKeys(MOODLE_PASSWORD_INCORRECT)
    // Submit the input field values
    loginButton.Submit()
    // Validate login
    let visibleElementWhenLoginSuccess =
            Moodle.driver.FindElementsById(itemVisibleWhenLoginSuccess)
    // Create a compound term to return
    if visibleElementWhenLoginSuccess.Count > 0 then
        Action.Create("login_finish", (t).[0], LoginStatus.Success)
                  :> CompoundTerm
    else
        Action.Create("login_finish", (t).[0], LoginStatus.Failure)
                  :> CompoundTerm
```

Figure 17. Login action in adapter

4 Test setup

The following chapter gives an overview of the server-side setup, generating load using model-based tests, using NModel metrics, PHP profiling and running Atop for Linux server performance analysis.

4.1 Server-side setup

The initial test setup consisted of two servers. A Linux server running Moodle with MySQL database and Apache web server running in the same machine. The specific Moodle server related parameters can be found under Appendix 2. A second server was used for generating load – running the model program instances. The two servers were connected with a secure shell (ssh) tunnel enabling sending and receiving http requests and responses. The motivation for using ssh tunnel for testing purposes relates mostly to web application security – making the Moodle test server not visible to the general public. There are some possible issues that might arise from such setup, namely if the processor load on the load generating server peaks then the test results may no longer be valid and may be caused by the bottleneck created by using the ssh tunnel. Figure 18 illustrates the test setup used.

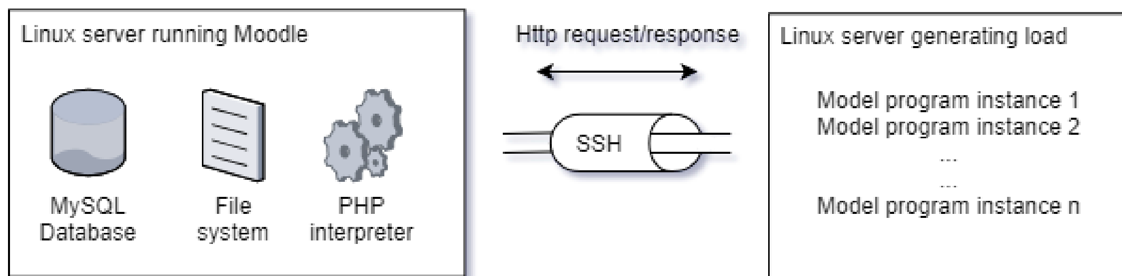

Figure 18. Test setup

4.2 NModel Metrics

NModel has the option of showing test-suite metrics when running the *ct*. To turn on metrics tracking, */metrics:+* must be specified in the *ct* argument file. When the test-suite has finished running the test metrics are printed in console [3]. Figure 19 shows an example of test metrics output.

```
Test Metrics(
    General Summary(
         30 tests Executed,
         1 tests Failed,
         Pass Rate: 96,7%
    ),
    Failed Actions(
         (logout_start, Reason: Action timed out( 1 time ))
    ),
    Toatal time spent in each action(
         quiz_start(Executed 4 times, Average execution time:
00:00:06.7680000) : 00:00:27.0756670,
         enrol_start(Executed 18 times, Average execution time:
00:00:04.1420000) : 00:01:14.5650120,
         login_start(Executed 54 times, Average execution time:
00:00:03.1480000) : 00:02:50.0181950,
         logout_start(Executed 29 times, Average execution time:
00:00:00.5520000) : 00:00:16.0299950,
         search_start(Executed 36 times, Average execution time:
00:00:03.2820000) : 00:01:58.1524460
    )
)
```

Figure 19. NModel metrics output

NModel metrics can be used to see the general impact of having a load on the Moodle server – it shows the increase in test execution time and the reasons behind test failure. MModel metrics was used to determine the upper limit of test users that could be used without getting more than a few action timeouts when running the tests.

4.3 Generating load using model-based tests

There were two options for generating load using the model created for this thesis. The first option was adding users to the model and therefore increasing the model complexity. That did not seem as the best option as the complexity and size of the model increases exponentially when adding more parameters. Figure 20 shows the size of the model with 4 users.

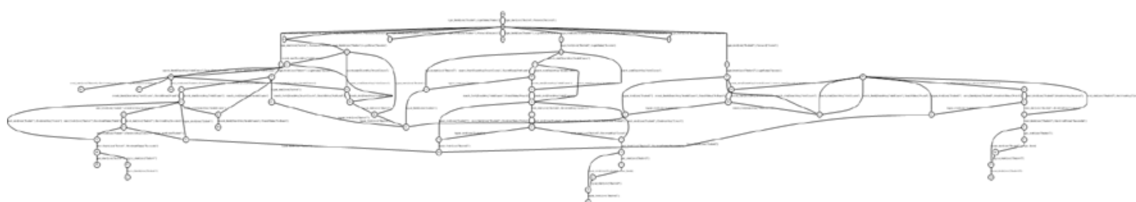

Figure 20. Model size with 4 users

The second option was running multiple instances of the same model in parallel. The result of that would be similar to the real life where users try to interact with different parts of the application simultaneously. The key to achieving that was getting multiple instances of the *ct* to run at the same time.

There were some obstacles in achieving that. Firstly, NModel did not support passing arguments to the stepper hence some changes had to be made to the NModel source code. A new option was added to the *ct* called *stepperArg* which takes an argument of the type of string and passes it the constructor of the stepper implementation. The stepper then checks if the argument is present and passes it on to the adapter. The argument passed on to the adapter was the username which enabled calling the tests with different users. Secondly, to start multiple instances of the *ct* a bash script was created. The script takes a newline separated file of strings with usernames, reads it in and then appends the usernames to the content of the original *ct* arguments file, saves the text to a new file and calls *ct* with the file as an argument.

Help from the supervisor, Juhan Ernits, was used in the process of overcoming the aforementioned obstacles. Figure 21 shows the bash script created.

```
#!/bin/bash
for i in  `cat args.txt` ; do
        cat ct_args_online.txt > ct_args.txt.$i
        echo "/stepperArg:\"$i\""  >> ct_args.txt.$i
        mono ./bin/ct.exe @ct_args.txt.$i &
done
```

Figure 21. Bash script to start multiple instances of the *ct*

4.4 PHP Profiling

Script profiling gives an idea which part of the code might not be performing well. Moodle documentation suggests using one of two PHP profilers – XHProf or XDebug. For this thesis, XDebug was chosen because of its general popularity and ease of use.

XDebug is a powerful profiling tool first released in 2002. It offers options such as function tracing, remote debugging and scripts profiling. The results from XDebug can be analysed using external tools like Webgrind, KCacheGrind or WinCacheGrind. To use XDebug, the extension must be installed and configured. There are a few options when it comes to enabling XDebug: keeping the profiler always enabled, using a trigger in form

of a cookie or sending a get/post request to start the profiling. Using a trigger is the preferable option as the profile data files are relatively large and triggering the profiling under specific conditions helps keep the amount of data generated under control [18].

To enable profiling, a web page that added a cookie named *XDEBUG_PROFILE* to the session was created. Profiling was started only for one specific user and the rules for that were defined under the adapter code.

For analysing XDebug profiling files Webgrind [19] was chosen. Webgrind enables tracking time spent in internal and user functions as well as function call tracing. Figure 22 shows a screenshot of the Webgrind UI. The following things can be seen:

- Invocation count – the number of times each function was called.
- Total self-cost – the total time it took to execute raw PHP in function.
- Total inclusive cost – total time, including other functions called.

Total inclusive cost in milliseconds will be the point of reference used in this thesis.

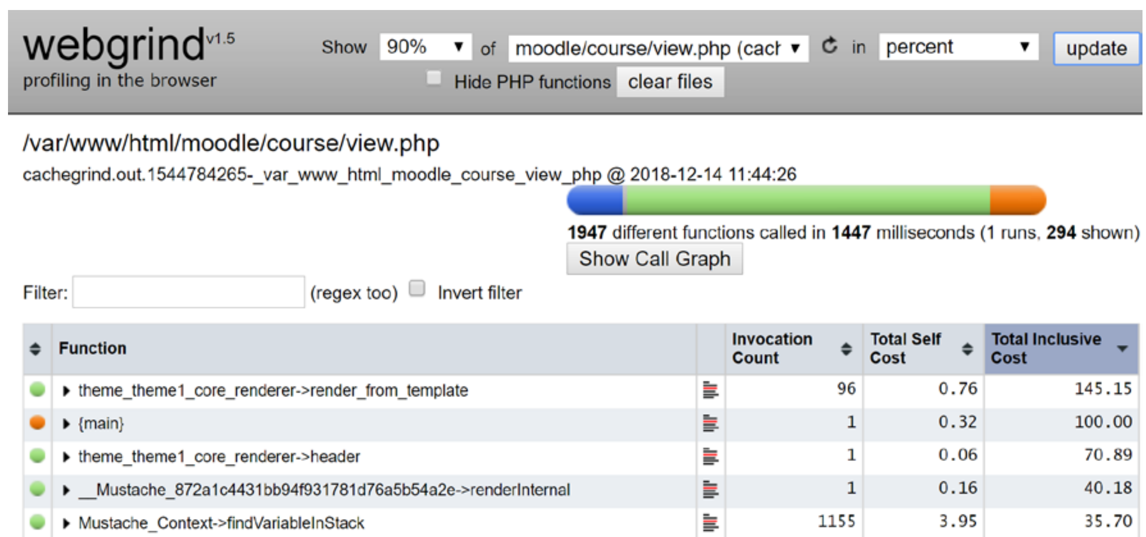

Figure 22. Screenshot of webgrind UI

4.5 Atop monitoring

Atop is an interactive monitor that can be used to view the load on a Linux system. It displays the occupation of hardware resources like CPU, memory, disk and network [20]. It has the option of saving the monitoring results to a raw file which can be opened and analysed later. Gnuplot was used together with Atop to visualise the monitoring results.

Gnuplot is a portable command-line driven graphing utility for Linux, Windows, OSX and other platforms [21]. In this thesis, CPU monitoring and disk busy percentage monitoring were used. For CPU monitoring an average with 1-minute interval was graphed for all the test cases. Disk busy percentage indicates the resource consumption on system level. For disk busy percentage monitoring, data was parsed from atop raw files and then graphed together with the most CPU heavy process.

In each test run Atop was started and results of monitoring saved into a separate log file. To start logging *atop -w file_name.raw* was called from the command line. The raw files were later graphed using a bash script and the results saved into a pdf file. Figure 23 illustrates the bash script that was created to graph Atop log files for CPU load information.

```
#!/bin/sh -u
#   $0 [atop logfile to plot]

log=${1-'/var/log/atop.log'}

tmp=/tmp/atop$$
rm -f $tmp
trap "rm -f $tmp" 0 1 2

atop -PCPL -r "$log" >$tmp

gnuplot -persist <<EOF

set xdata time
set timefmt '%Y/%m/%d %H:%M:%S'
set format x "%Y\n%m/%d\n%H:%M"
set grid
set key right top
set title "CPU load"
set terminal pdf
set output "plot_cpu_1.pdf"
plot \
   "$tmp" using 4:8 t ''  , \
   "$tmp" using 4:8 smooth csplines t 'CPU load avg1'  ;
EOF
```

Figure 23. Bash script for graphing Atop log files

5 Test results

The following chapter gives an overview about the test results when running the tests with 30 and 40 users and comparing that to the results of single user test.

5.1 NModel metrics

The first point in observing the test results was looking at NModel metrics. For that, after each test run (1 user, 30 users, 40 users) the time it took to execute each action in the model was noted and for multi-user tests an average of execution times was calculated in seconds. Each test suite consisted of 30 test runs for each user enabling to make sure that every action in the model was called and to get a good average of test execution times per user.

The one user scenario gives a good reference point to the response times when there is little to no load on the system. The 30-user scenario which already had a few test failures due to timeouts (web page taking more than 5 seconds to load) gives a benchmark on how the system acts under moderate load. As can be seen from the Table 4, the increase in test run time was not too noticeable but the impact of having 30 users use the system simultaneously already caused the occasional timeouts. The most noticeable increase being in the Enrol and Search actions (~0.5s and ~0.7s). In the case of 40 users the amount of test failures increased noticeably – on average 2 out of the 30 runs failed. The increase in action times was following: Quiz (~1.5s), Enrol (~1.5s), Login (~1s), Logout (~0.3s) and Search (~1.1s). The 40-user scenario works as a benchmark for high load on system – the load being enough to have timeouts happen regularly but not enough to cripple the system completely. It should be noted that the test failures tended to happen in clusters with several failures in a row.

	Test run count	Failed	Quiz (s)	Enrol (s)	Login (s)	Logout (s)	Search (s)
1 user	30	0	7.218	3.245	3.505	0.662	2.374
30 users avg.	30	0.267	7.232	3.768	3.575	0.603	3.027
40 users avg.	30	2.025	8.7	4.778	4.537	0.92	3.424

Table 4. NModel metrics time spent in each action in seconds.

5.2 Atop monitoring

The results from NModel were coupled with Atop monitoring. For each test suite, Atop monitoring was run and log files saved giving valuable information about what was happening with the CPU load and disk usage. The logs from Atop were later turned into graphs using Gnuplot and Excel.

Figure 24 illustrates the average load on the Moodle server when 1 user was using the system. As it can be seen, there is a relatively small increase in the CPU average load compared to when the system was idle. On the horizontal axis is time and on the vertical axis there is CPU 1-minute load average (the same applies to all graphs illustrating CPU load average). There are 4 CPU cores on the test machine meaning that when CPU average load is above the number of CPU cores there is a high demand for the CPUs and when it's below then the CPUs are underutilized [22].

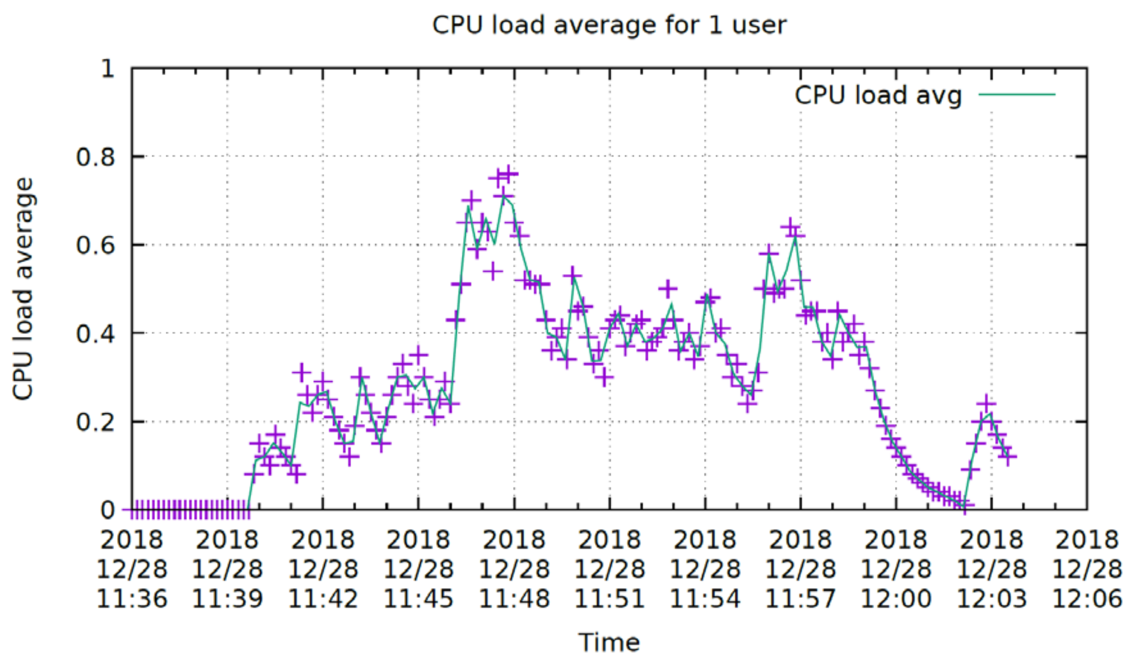

Figure 24. CPU load average for 1 user

Looking at the raw atop log files, it can be seen that the most CPU intensive processes in that test suite were apache2 and MySQL. As can be seen on Figure 25 the disk busy percentage remains low and so does the MySQL CPU usage percentage. On the x axis is time and on the y axis there is resource utilization. The highest utilization for CPU usage can be 400% as the system is running on 4 cores.

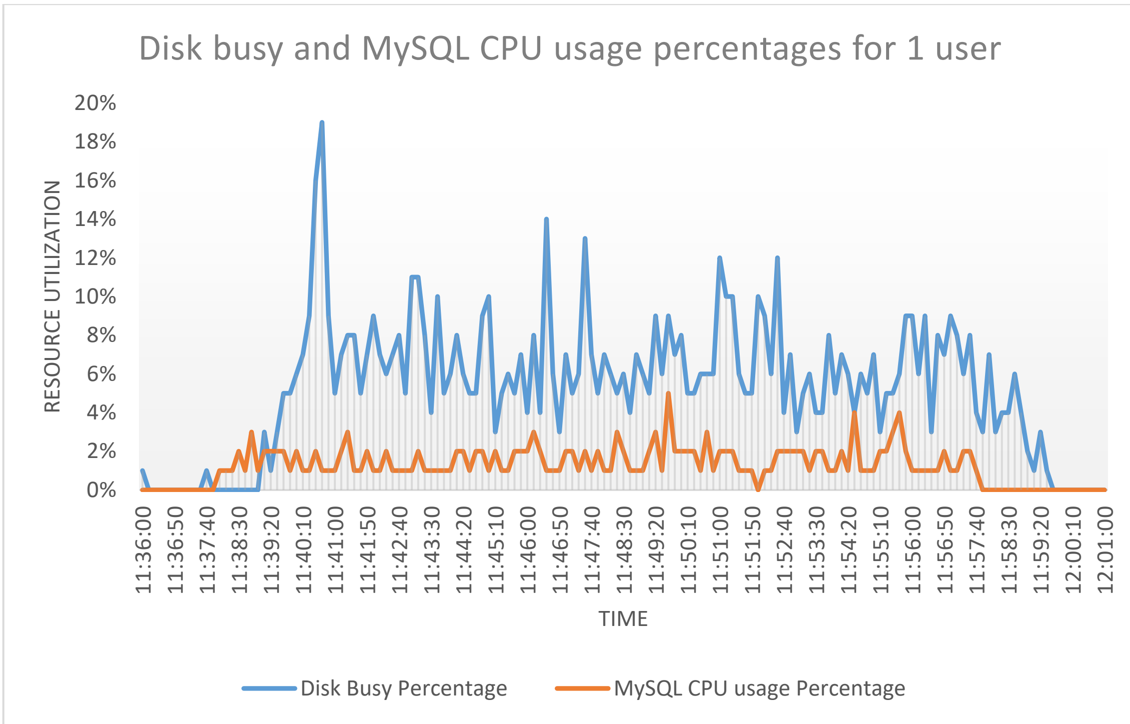

Figure 25. Disk busy and MySQL CPU usage percentages for 1 user.

Figure 26 shows how the system performed under moderate load with 30 users. It can be noted that the load during testing was quite stable.

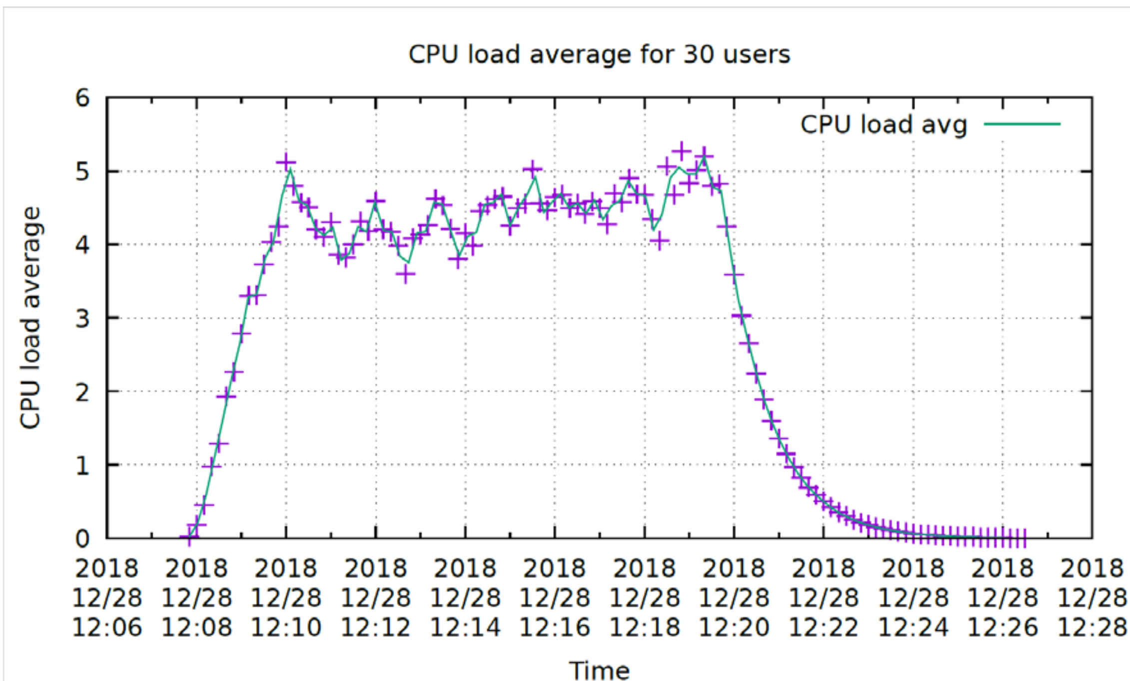

Figure 26. CPU load average for 30 users

While the CPU average load remained relatively low, it can be seen on Figure 27 that the disk busy percentage is around 35-40% on average with peaks reaching towards 60%. It should be acknowledged that disk busy load over 70% is considered critical explaining the occasional timeouts when the tests ran. Compared to the test suite with one user, the most CPU intensive processes are MySQL and SSHD. A correlation between the MySQL CPU usage and disk busy percentages can be seen. An increase in CPU usage for the SSHD (from 1-2% in case of 1 user to 10-15% with 30 users) process could be observed as well when looking at the atop logs. SSHD is a process that listens to incoming connections using the SSH protocol and channels the queries to the web server from the tester and returns the queried web pages. Increase in CPU load for the SSHD process in in correlation with increased web traffic.

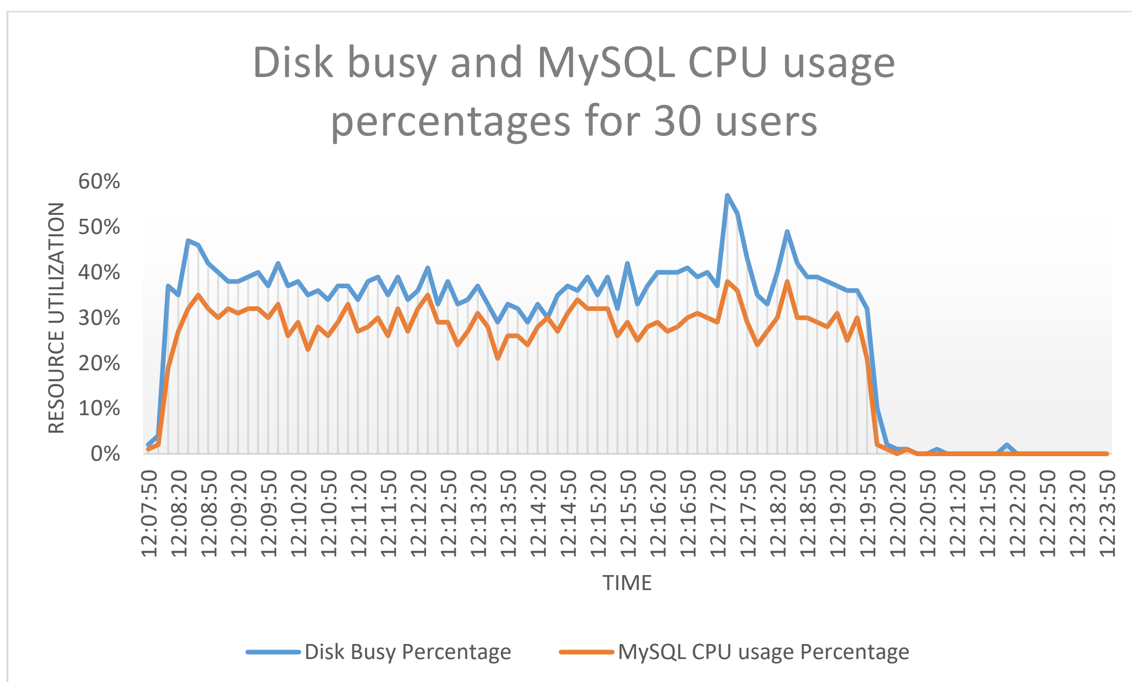

Figure 27. Disk busy and MySQL CPU usage percentages for 30 users

Figure 28 illustrates the average CPU load for 40 users. Compared to the 30-user test suite the load has increased more than three times. There is a noticeable increase in CPU load around 10:30. When comparing the CPU load graph to Figure 29 disk busy percentage there is a peak in disk busy percentage around the same time.

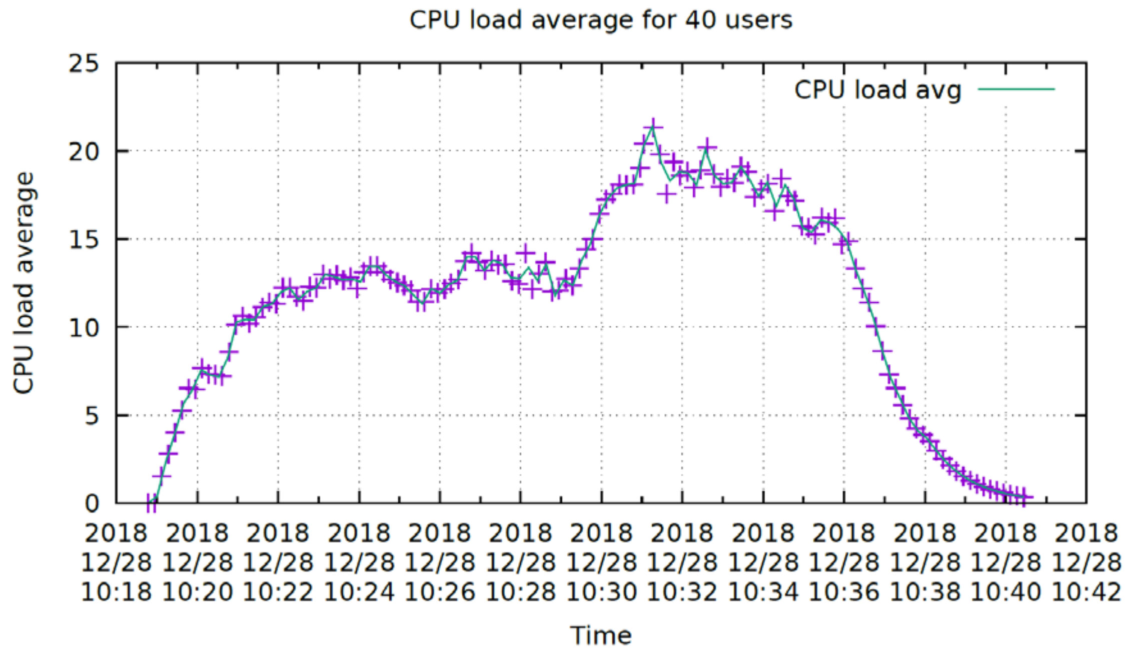

Figure 28. CPU load average for 40 users

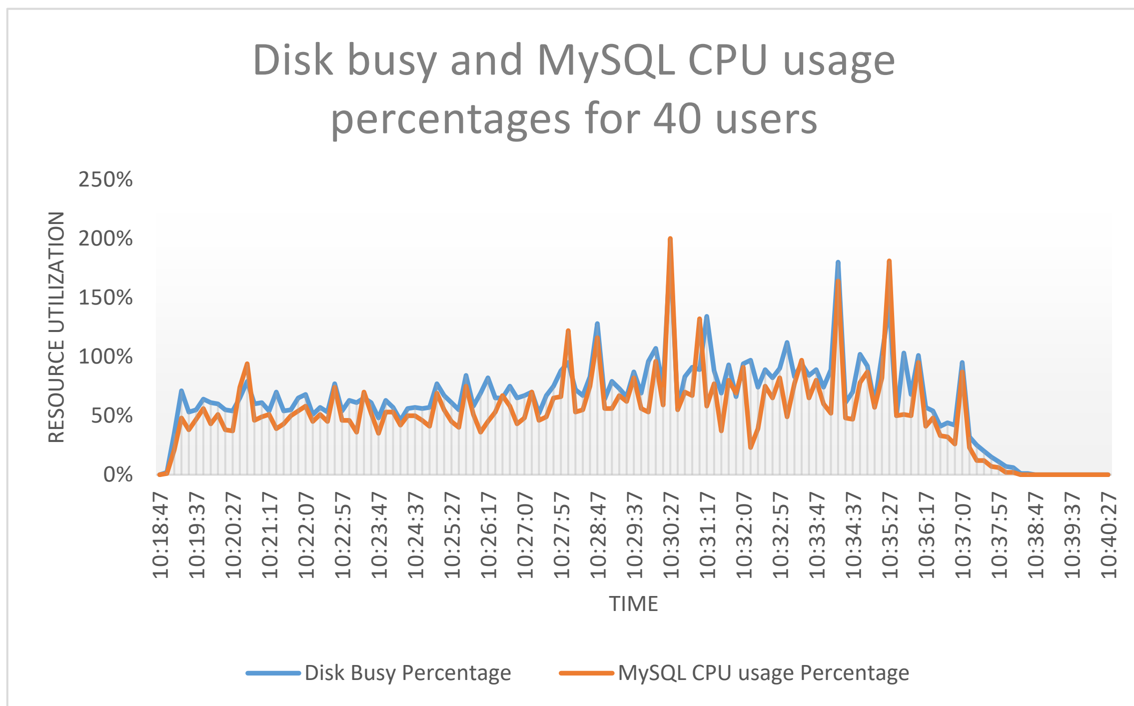

Figure 29. Disk busy and MySQL CPU usage percentages for 40 users

Figure 29 shows disk busy percentage and MySQL CPU usage percentages. There is a strong correlation between the two. Disk busy percentages in case of 40 users go as high as 200% (above 70% is considered critical load). One possible explanation for the increase in CPU usage could be that the CPU(s) are waiting for an ongoing disk write/read

(I/O). While Atop does have the option of looking into the processes that are using disk I/O, the option was not enabled in the test server and therefore more detailed information on disk usage was not available. Nevertheless, it can be inferred that as disk is working under critical load most of the time in case of 40 users, most probably the bottleneck lies in the disk usage.

5.3 PHP Profiling

Each test suite was run with a specific test user for whom PHP profiling using XDebug was started by adding a cookie to the browser session. The profiling logs were later analysed and visualized using Webgrind. The profiling worked for 1-user and 30-user scenarios but not for 40-user scenario. For some reason, log files were not created for 40-user scenario even after several runs. One possible explanation could be that the load on the disk somehow affected writing logs.

Even though profiling did not work in the 40-user scenario, comparing only 1-user and 30-user scenarios gives an idea about what was happening with the PHP functions and the time it took to call them. The comparisons were done on the course home page (../course/view.php).

Looking at Figure 30 profiling results for 1 user and Figure 31 profiling results for 30 users an increase in total inclusive cost (in ms) can be seen. Further analysis of the logs reveals that the increase cumulated from the following areas:

- Time it took to execute database queries.
- Time it took to fetch files from the server.
- Time it took to render web page templates.

The first two could be related to the high load on disk directly and the third one indirectly as other processes might get influenced by the high load on disk.

Function		Invocation Count	Total Self Cost	Total Inclusive Cost
theme_theme1_core_renderer->render_from_template		72	9	1645
{main}		1	5	1470
theme_theme1_core_renderer->header		1	1	956
include::/var/www/html/moodle/theme/theme1/layout/course.php		1	1	940
__Mustache_872a1c4431bb94f931781d76a5b54a2e->renderInternal		1	1	473

Figure 30. Profiling results for 1 user

Function		Invocation Count	Total Self Cost	Total Inclusive Cost
theme_theme1_core_renderer->render_from_template		308	43	3632
{main}		1	4	3426
format_weeks_renderer->print_multiple_section_page		1	7	1958
theme_theme1_core_renderer->render		150	16	1466
theme_theme1_core_course_renderer->course_section_add_cm_control		5	8	1265

Figure 31. Profiling results for 30 users

5.4 Assessment of model-based load testing

Using model-based testing to conduct load testing was quite challenging. The main challenges being the time it took to implement the model-based tests, getting multiple instances of tests to run at the same time and making sense of the results. Despite the challenges, model-based testing using NModel seemed to work for the purposes of load testing. Lifelike load was generated on the Moodle web application and possible bottleneck found by combining different tools. The benefit of using model-based load testing could be that a single instance of tests could be used as part of regression/general testing as well as for load testing using only some specific areas of the model.

There are some areas for improvement though. Firstly, running NModel with multiple instances required some “hacking” which could become an obstacle were other people to use the same setup. Secondly, NModel metrics could be improved upon as right now gathering the data from metrics meant going through the whole NModel log file and manually copying the data. One possible suggestion to improve that could be saving metrics logs to a separate file in a more easily parsable format. Thirdly, perhaps more consideration should be put into logging and graphing tools by choosing a combination that has out-of-the-box graphing utilities preferably with a web page interface as there were considerable difficulties in making sense of the log files and later graphing them.

Summary

The purpose of this thesis was to find out how model-based load testing using NModel framework could be used to detect the performance issues affecting Moodle web application. For that, requirements of the functionality under test were gathered and then modelled using NModel. A test harness was created together with a system adapter connecting the model to the system under test. Some changes were made to the NModel source code to enable passing an argument to the Stepper in order to run multiple instances of tests simultaneously. PHP profiling using XDebug was set up on the server-side together with Webgrind – a tool to analyse the profiling results. Atop monitoring was run on the server together with the test suites to see what was happening to the server-side performance. Combining the results from NModel metrics, Atop logging files and XDebug profiling logs it could be speculated that the problem behind the performance issues of Moodle web application lies in the hard drive. Disk busy percentages in high load tests were beyond critical level.

In general, using NModel for load testing does work with some issues. Using a tool designed specifically for load testing would perhaps give a better overview and observability to what is happening in the tests as well as more easily understandable test results. At the same time, using model-based tests for the purposes of load testing does give a more lifelike simulation of user-behaviour having multiple users doing different actions on the system at the same time.

The proposed approach is fit for modifying the system under test, e.g. by changing configuration, and then re-running the tests to see whether the tests generated in this thesis could be used to find new bottlenecks. Additionally, some improvements to the NModel tool can be suggested. For example, adding timestamps to log files and saving metrics logs separate from general logs as well as creating a more straightforward way to run multiple instances of the tests at the same time.

References

[1] E. Tkachenko, “Testing the Requirements: A Guide to Requirements Analysis,” 15 Feburary 2018. [Online]. Available: https://www.techwell.com/techwell-insights/2018/02/testing-requirements-guide-requirements-analysis. [Accessed December 2018].

[2] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux and A. Vernotte, “Recent Advances in Model-Based Testing,” *Advances in Computers,* vol. 101, pp. 53-120, 2016.

[3] C. Campbell, J. Jacky, M. Veanes and W. Schulte, Model-Based Software Testing and Analysis with C#, Cambridge University Press, 2007.

[4] D. Desai, V. Gautam and D. S. Dhaka, “Model-Based Testing,” 2018.

[5] “Model Based Testing,” 2016. [Online]. Available: http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Model_Based_Testing. [Accessed December 2018].

[6] G. Hombrebuen, “How Model-Based Testing Tools Can Help Ease MBT Adoption,” 9 February 2018. [Online]. Available: https://www.ca.com/en/blog-agile-requirements-designer/how-model-based-testing-tools-can-break-barriers-to-mbt.html. [Accessed December 2018].

[7] C. Mandrioli, M. Jivung and C. J. Balck, “Model Based Testing,” 2018.

[8] X. Wang, B. Zhou and W. Li, “Model Based Load Testing of Web Applications,” in *International Symposium on Parallel and Distributed Processing with Applications*, 2010.

[9] J. Zhou, B. Zhou and S. Li, “LTF: A Model-based Load Testing Framework for Web Applications,” in *14th International Conference on Quality Software*, 2014.

[10] “NModel: Library for model-based testing in C# (and F#),” [Online]. Available: https://github.com/juhan/NModel. [Accessed December 2018].

[11] G. Kolawole, “Model based testing mobile applications: a case study of moodle mobile application,” 2017.

[12] J. Ernits, “Model-Based Testing with NModel,” [Online]. Available: http://193.40.251.102/tiki-download_wiki_attachment.php?attId=322. [Accessed December 2018].

[13] “About Moodle,” [Online]. Available: https://docs.moodle.org/36/en/About_Moodle. [Accessed December 2018].

[14] “Apache JMeter™,” [Online]. Available: http://jmeter.apache.org/. [Accessed December 2018].

[15] T. Hunt, “Load-testing Moodle 2.6.2 at the OU,” 25 April 2014. [Online]. Available: https://tjhunt.blogspot.com/2014/04/load-testing-moodle-262-at-ou.html. [Accessed December 2018].

[16] “Standard roles,” [Online]. Available: https://docs.moodle.org/36/en/Standard_roles. [Accessed December 2018].

[17] A. Jones, “Behavior Driven Testing in Automated Testing,” [Online]. Available: https://dzone.com/articles/behavior-driven-testing-in-automated-testing-2. [Accessed December 2018].

[18] “Profiling PHP,” [Online]. Available: https://docs.moodle.org/dev/Profiling_PHP. [Accessed December 2018].

[19] “Webgrind,” [Online]. Available: https://github.com/jokkedk/webgrind. [Accessed December 2018].

[20] “atop(1) - Linux man page,” [Online]. Available: https://linux.die.net/man/1/atop. [Accessed 1 January 2019].

[21] “gnuplot homepage,” [Online]. Available: http://www.gnuplot.info/. [Accessed 1 January 2019].

Appendix 1 – Source code to the Moodle model and adapter

The source code to the Moodle model and adapter can be found at https://github.com/pawzy/MoodleTest.

Appendix 2 – Server parameters

Ubuntu 18.04, x86_64

RAM: 8 GB

Swap space: 4GB

4 processor cores (hyperthreading enabled)

Processor: Intel(R) Xeon(R) CPU E5645 @ 2.40GHz

Virtualization: KVM/Qemu virtualization, Linux kernel 4.15

Disk virtualization: native ZFS disk dataset

Network virtualization: emulated E1000 network interface

Database: Mysql version 5.7.24, default parameters.