TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Gabriel Kolawole IVSM155847

MODEL BASED TESTING MOBILE APPLICATIONS: A CASE STUDY OF MOODLE MOBILE APPLICATION

Master's Thesis

Supervisor: Juhan-Peep Ernits, PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tarkvarateaduse instituut

Gabriel Kolawole IVSM155847

MOBIILIRAKENDUSTE MUDELIPÕHINE TESTIMINE MOODLE' MOBIILIRAKENDUSE NÄITEL

Magistritöö

Juhendaja: Juhan-Peep Ernits, PhD

Tallinn 2017

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: Kolawole Gabriel

Abstract

The tools for the development of mobile applications are in rapid development, with different devices and mobile operating systems that are being introduced in the market periodically, this poses a need to have an easy and effective approach of testing mobile applications in a cost effective manner, one of such approaches is testing these applications based on the way they are expected to behave with different range of inputs and scenarios which is termed Model Based Testing (MBT).

The current thesis describes the process of performing MBT on the Moodle educational software mobile application. Moodle Mobile app is the official front end to Moodle and is available for Android and IOS.

This thesis has presented a behavioral approach in testing mobile applications using MBT with which an application can be tested with multiple test cases on the fly or offline. This was achieved by developing an adapter between the application behavior models and a real device using JSON protocol exposed by Appium.

The first part of this thesis introduces the key concepts of MBT . The second section is centered on the state of the art and the related work. The third section discusses model programs developed in this work using NModel which is the framework which is used for modelling. The fourth section focuses on tools setup and entire test architecture. Section 5 detailed the adapter implementation and the end to end process of achieving the MBT using NModel.

This thesis is written in English and is 54 pages long, including 5 chapters, 14 figures and 2 tables.

Annotatsioon

Mobiilirakenduste arendamise vahendeid arendatakse pidevalt edasi. Erinevate seadmete ja mobiilsete operatsioonisüsteemide versioonide perioodilise lisandumisega turule suureneb vajadus lihtsa ja efektiivse lähenemise järele, mis võimaldaks mobiilirakendusi testida kuluefektiivselt. Üks selliseid lähenemisi on kasutades käitumislikku lähenemist erinevate sisenditega ning stsenaariumitega, mida võimaldab mudelipõhine testimine.

Käesolevas magistritöös kirjeldatakse, kuidas mudelipõhiselt testida Moodle mobiilirakendust. Moodle Mobile App nimeline mobiilirakendus on Moodle ametlik kasutajaliides mobiilsetele seadmetele, mis on kättesaadav nii Androidile kui ka IOS-le.

Antud magistritöös tutvustatakse mobiilirakenduste käitumuslikku testimist kasutades mudelipõhist testimist erinevate testijuhtude käivitamiseks eelsalvestatud testidest või mudelit käigupealt interpreteerides. See saavutati adapteri arendamisega, mis võimaldab käitumislikke mudeleid käivitades juhtida mobiilirakendust Appiumi kaudu üle JSON keeles defineeritud protokolli.

Magistritöö esimeses osas tehakse sissejuhatus mudelipõhise testimise põhimõistetele. Teises osas keskendutakse mobiilirakenduste testimisvahenditele ja mudelipõhiste testimislahenduste arengutele. Kolmandas osas tutvustatakse mudelprogrammide loomist NModel teeki kasutades. Neljandas peatükis keskendutakse kogu testimislahenduse ülesseadmise detailidele. Viiendas peatükis kirjeldatakse Moodle Mobile Appi jaoks adapteri loomist ja NModeliga selle rakenduse testimise protsessile.

Käesolev magistritöö on kirjutatud inglise keeles 54-1 leheküljel, sisaldades 5 peatükki, 14 joonist ning 2 tabelit.

List of abbreviations and terms

SUT	System Under Test
MGT	Model based GUI Testing
AUT	Application Under Test
FSM	Finite State Machine
API	Application Programming Interface
MBT	Model Based Testing
GUI	Graphical User Interface
UI	User interface
FSM	Finite State Machine
SRS	System Requirement Specification
APK	Android Application Package

Table of contents

List of figures

List of tables

1 Introduction

Mobile application development and software development has evolved over the years with the advent of various tools and methodologies for designing and implementing working solutions on both mobile and desktop systems. With every tool, IDE, library involved in the process of crafting the application some additional complexity and bugs may build up given that some of such features can be poorly documented.

The usual workflow follows the V-Model of software design in each iteration where an architect or a product owner, conceptualizes a solution to a problem, which is then transferred into words as requirements which are then processed into detailed designs, these designs are then implemented. On a successful implementation, the system goes through the testing phase where series of tests are performed such as unit, integration and system level tests.

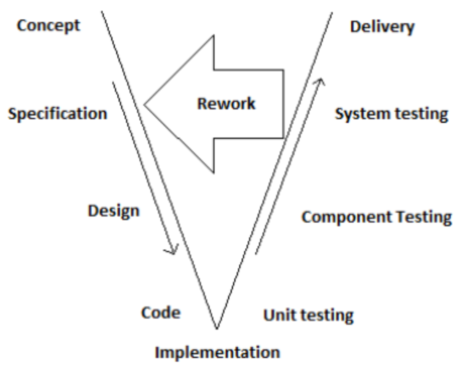


Figure 1 V-Model Diagram [1]

Mobile applications are conceptualized in the form of wireframes and sketches which forms the working input for the development team to provide a working solution to each requirement as stated in the requirement document. In practice the developer or engineer develops a working solution for which often a set of unit tests gets developed which is usually dependent on the underlying language used in developing the solution.
Unit testing is the most 'micro' scale of testing; to test functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code [2]. Not always easily done unless the application has a well-designed

architecture with a very effective source code which get a task done with minimum resource, may require developing test driver modules or test harnesses [3].

The next move from unit testing is to test various components of the system through integration tests. Integration test reveals defects of interfaces and integrated components or systems interactions [4]. With integration testing communication, related defects with various participating components are be detected. To achieve delivery of the application, system testing is done, system test in involves user acceptance during which each of the requirements for the solution are validated, this is performed based on the SRS, system testing is performed on a complete and fully integrated software product, it is the final test to be conducted where both functional and non-functional requirements are investigated [5].

Mobile application usage and development is experiencing exponential growth. According to Gartner, by 2016 more than 300 billion applications will be downloaded annually. The mobile domain presents new challenges to software engineering. Mobile platforms are rapidly changing, including diverse capabilities as GPS, sensors, and input modes. Applications must be omni-channel and work on all platforms. Developing such applications requires suitable practices and tools e.g., architecture techniques that relate to the complexity at hand; improved refactoring tools for hybrid applications using dynamic languages and polyglot development and applications; and testing techniques for applications that run on different devices [6].

Software testing has grown as an important technique to evaluate and help to improve software quality. Numerous techniques and tools have appeared in the last decade, ranging from static analysis to automatic test generation and application [7]. A perfect world for mobile application testing will be a scenario where a single build of the application can be used to test and simulate user/server interaction with the application with zero human intervention in terms of interaction for testing. The test will be driven using pre-written test drivers which can simulate user interaction by sending commands through some interface between the mobile device and the underlying application running on the device. In this case, the tests will usually aim at asserting if equal application states are reached for the equivalent commands sent to the device.

This thesis will describe the complete flow of testing Android applications in a behavioral approach using NModel which is an MBT framework and Appium which is an automation framework for Android and IOS. This work will provide a complete tool chain which is required in running model based tests on Android devices. A complete solution containing various system adapters, scenario FSM's and all other relevant source codes is also delivered.

Model Based Testing

Model based testing (MBT) is software testing using programs (models) modelled to behave as the functionalities or requirements of the system under test (SUT), this forms the base of generating test cases which drives the testing process. The functions of the system under test (SUT) are abstracted at a very high level to achieve a very high level of automation, and this can easily be applied and implemented during system and integration test of the SUT. The verdicts from MBT can easily be used to rework the system so that a suitable solution is delivered.

During component testing, different components of the system can be modelled as features of the overall system which can be combined to test the overall system. The advantage of testing the system using this approach includes:

i. Meaningful test cases can easily be generated
ii. Bugs of the system can be detected earlier.
iii. Reduced the cost of maintenance. You do not need to write new tests for each new feature. Once you have a model it is easy to generate and re-generate test cases than it is with hand-coded test cases [8].
iv. Test cases can easily be generated from large state machines by specifying relevant scenario
v. Modelling help to refine unclear and poorly defined requirements [9].

MBT can be achieved with either online or offline approach. Online MBT involves the generation of test cases in motion, which is useful in testing nondeterministic systems. In this case, test cases can have very large number of steps.

Offline MBT involves automating test case generation where test cases are fixed at the time of test generation and execution of tests is performed later.

A typical workflow for MBT is as follows:

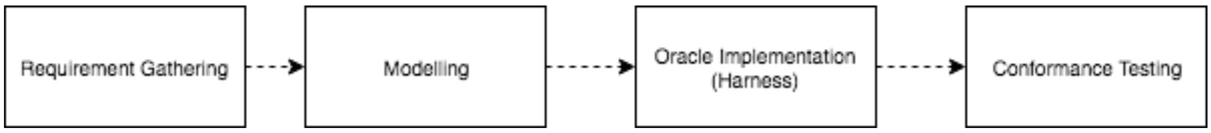


Figure 2 MBT work flow

1.2 Modelling

This is the process of describing the SUT requirements which will be used by the test generator, for generating test cases, as well as the output expected from the SUT. This involves a very high level of abstraction of the SUT functions.

The objective during the modelling process is to be able to figure out which functions of the SUT are to be tested, as well as deciding on variables or data structure that best describe the SUT.

Every action of the model program to be modeled (written) must be able to answer the following:

- Action enabling: a proper condition must guard the enabling of each action.
- State Update: which state variable or entity will be updated?
- Return value: it is must be determined whether the action returns a value or not.

1.3 Model Programs

A model program is a simple program describing the behavior and expected outputs of the system under test. Where every feature of the specification is a action method in the model program covering controllable such as input entry, button clicks and uncontrollable user actions like application state changes, sever responses in a client-server scenario can all be captured within a model program. The model is the test case generator and oracle [10].

A model program abstracts the implementation of the SUT at a very high level. Important terms related to model program;

Actions - This is a unit of behavior of the SUT. An action could be a single activity or comprise of other activities. Action represents every method in the implementation of the SUT. Model program's actions are decorated like *[Action]*.

States - This is the situation of things in the SUT at a point in time. At every point in system variables takes achieve new state / values. In the model program this are represented as variables which does not have to correspond to the variables in the implementation of the SUT. Execution of a SUT begins from an initial/idle state to an accepting state where the goals of the system have been achieved. A series of actions in sequence from initial state to the accepting state is called a *run/trace.*

Behavior - This is the complete collection of runs/traces the SUT can attain.

Contract Model Program - This specifies all permissible behaviors of the implementation of the SUT. This is the specification of the dos and don'ts of the implementation of the SUT.

Scenario – A specific subset of all possible runs/traces that adhere to a specific scenario.

A *Scenario model program* is peculiar to the scenario described in the form of a model program. It is intended to be composed with the contract model program to restrict the set of behaviors to a specific scenario.

A typical workflow for MBT is the following.

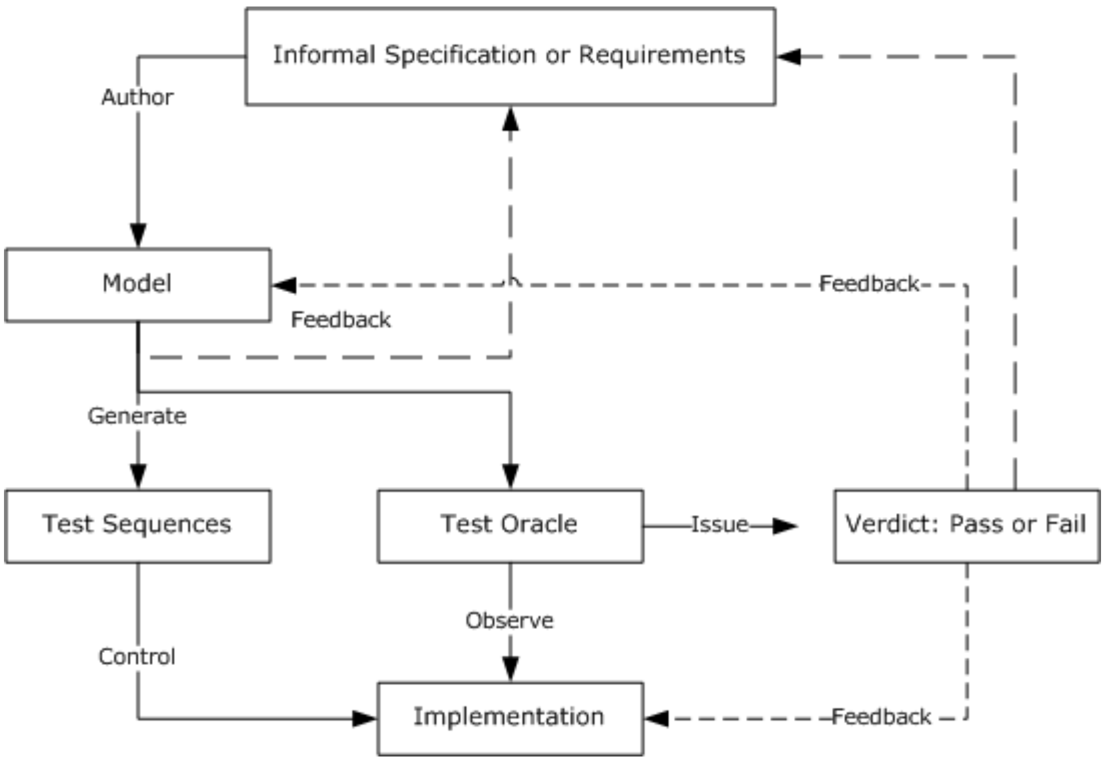


Figure 3 Typical Model based testing setup [2].

1.4 Aims and Objectives

The aim of this thesis is to develop and demonstrate the process of model based testing with focus on behavior/ scenario control approach for mobile applications using model programs written with NModel framework.

My work will cover the following:

- Setting up different tools/frameworks required in achieving MBT testing
- How to implement a system adapter which will be used to glue model programs which models a system behavior to the actual system execution.
- Writing model programs for the case study of this thesis, i.e. formalizing the requirements.
- Testing of the case study application and reporting of the test results.

2 Background and Related Work

State of the art of mobile application testing.

In a world where mobile applications are getting more sophisticated, relying on data from various sources (agents) to ensure their smooth operation, automating the testing of most significant behaviors of such system is crucial in quality control.

Therefore, to ensure the reliability of the software, along with quality assurance, verification and validation, researchers have developed several techniques for the different abstraction levels of software testing, i.e., unit, integration and system [10].

Automated Graphical User Interface (GUI) testing is one of the most widely used techniques to detect faults in mobile applications (apps) and to test functionality and usability [11].

The evolution of various tools and languages used in developing mobile application results into many challenges being encountered in testing the usability of this application. The challenges are not limited to the following:

- Diversity of operating systems,
- Accelerated development,
- Integration / collaboration with existing system,
- Nature of the application (native or hybrid),
- Internet connection.

Android applications are tested in various ways. And the strategy involved in this test is usually a function of the above stated challenges. Thus, test automation frameworks are introduced to test mobile applications.

Popular open source test automation test tools include Appium [12], Monkey Talk [13], Calaba.sh [14], Robotium [15], Selendroid [16]. What is common to all these tools is that they are plainly for UI acceptance testing with no consideration for depending systems/hosts or application usage scenario, this is which is not featured as to the usage and implementation of this tools.

MonkeyTalk is an open source test automation tool which can execute a test script on either an emulator or a real device, it supports Android and IOS. Monkey Talk like Appium works a mobile application build by executing a test script on against this build on a real device or an emulator. MonkeyTalk can perform the following, during execution: '*select, enterText, verify, entertext;* Using each of the commands.

To formulate a test, script a sequence of steps can be recorded using the monkeytalk ide, the result of the recording step a sequence of steps which can be replayed during automation, and this is what makes up the record and replay mode of monkey talk. Data driven testing is also possible with monkey talk where a set of inputs are pre-defined and these values are used by monkey talk during playback.

Another interesting testing approach is the Automated functionality testing through GUIs [17]. This is approach uses an MGT framework called the Action-Event framework (AEF). This framework helps testers abstract away from low-level details of the GUI under test and generate test cases in a behavior-oriented way. In this framework, testers can perform both business logic testing and GUI testing in a reusable manner. At the core of AEF is a mapping language that allows test engineers to map abstract actions to GUI implementations [17].

Taking mobile application testing beyond GUI acceptance testing, the behavioral approach can be used, where by a requirement of the system is given in a more declarative manner including an actor (feature) and what it is acting on as well as the expected result (output) of the action. For example, consider an inventory system where the statement below can be given as a feature:

- *Refunded or replaced items should be returned to stock* [18].

In the statement above the behavior of the system is expected to be that when customers are refunded for items, such items are expected to be seen in the stock and like wisely for when they are replaced. From the example given we can abstract this statement into a model, which is modelled using a modelling framework and from this model test cases covering different scenarios(behavior) of the system(model) are automatically generated with a test case generator provided by the modelling framework.

There are number of modelling frameworks which can be employed for this, they are discussed below;

2.1 Modelling Frameworks

2.1.1 Microsoft Spec Explorer

Spec Explorer is an MBT tool which extends the functionality of Visual Studio, software can be modelled, analyzed and visualized graphically. Spec explorer supports code generation, such that standalone code is generated from models.

The models can be written in C#, and test generation is directed with test purposes written in the Cord language. The right level of abstraction for the model used for test generation must be chosen by the engineer to keep the model manageable, which requires a deep knowledge on modeling and the tool [19].
Synchronous, asynchronous as well as non-deterministic systems can be modelled using spec explorer. Explored behaviors of a model are displayed as exploration graphs.

Spec explorer concepts are closely related to NModel, for instance the concept of *Action* is similar as this represents an interaction with the SUT in both NModel and spec explorer. Spec explorer also uses *Test Cases* & *Test Suites* concepts as available in NModel.

2.1.2 NModel

NModel is an open source modelling framework built on the experiences of Microsoft Spec Explorer [10]. User writes model programs in C#. NModel generates tests from your model program in advance or on-the-fly as the test run executes.
Users may add custom test generation strategies in C# to achieve any coverage criteria. SUT not limited to Windows [20].

2.1.3 PyModel

PyModel is an open source framework for model based testing in python. It is like the NModel as it was influenced by NModel. Internal concepts the framework implements e.g *composition,* it supports *offline* and *on-the-fly* testing. Composition is used for *scenario* control, where coverage is being guided by a programmable *strategy*. It provides the PyModel Analyzer *pma*, the PyModel Graphics program pmg for visualizing the analyzer output, and the PyModel Tester pmt for generating, executing, and checking tests, both offline and on-the-fly. It also includes several demonstration samples, each including a contract model program, scenario machines, and a test harness [20].

2.1.4 ModelJUnit

Model JUnit is an extension of the unit testing library in Java, (Junit) with the sole purpose of model based testing. ModelJUnit allows java classes to be used in creating   simple finite state machine (FSM) models or extended finite state machine (EFSM), from this tests cases can be generated and measure various model coverage metrics.

2.1.5 Conformiq's Qtronic

Qtronic is a commercial model testing tool developed by Conformiq Software Ltd. Like every other alternative, tests are driven from behavioral state models State space analysis of behavior that is implied by design model are used in generating test cases. Conformiq also generates automatic test case documentation in any language, which is uploaded to any test management system. When the SUT design changes Conformiq automatically update test scripts and test cases.
Qtronic is particularly suitable for component, sub-system, and system level testing [19].

Overall there is no much difference between these tools, as they influence each other regarding implementations, except for Conformiq which is commercial.

All model programs in this work will be done using the NModel and F#. The reason for using this two is because F# is a powerful functional language targeted for .NET framework and NModel is a choice because it is open source and intended to be improved during this thesis. The unique thing here is that when modelling with NModel we can take advantage of the various types which F# has to offer like Tuples, Discriminated Unions, Records and Enums.

Related literatures which are related to this work are describe below;

2.2 Web Applications Testing Using NModel [10].

Ernits J., Roo R., Jacky J., Veanes M. applied MBT in testing web applications in [4]. The SUT was a commercial web based positioning system called Workforce Management (WFM). The application tracks positions of employees. The application interacts with other system (services) such as the billing and positioning system.

Models of for the SUT was written in c#, and the online testing strategy was used, such that tests were generated on the fly.

The goal of this work was test the functional correctness of the WFM under normal working conditions as well as startup and shutdown of the system.

The test setup is as follows:

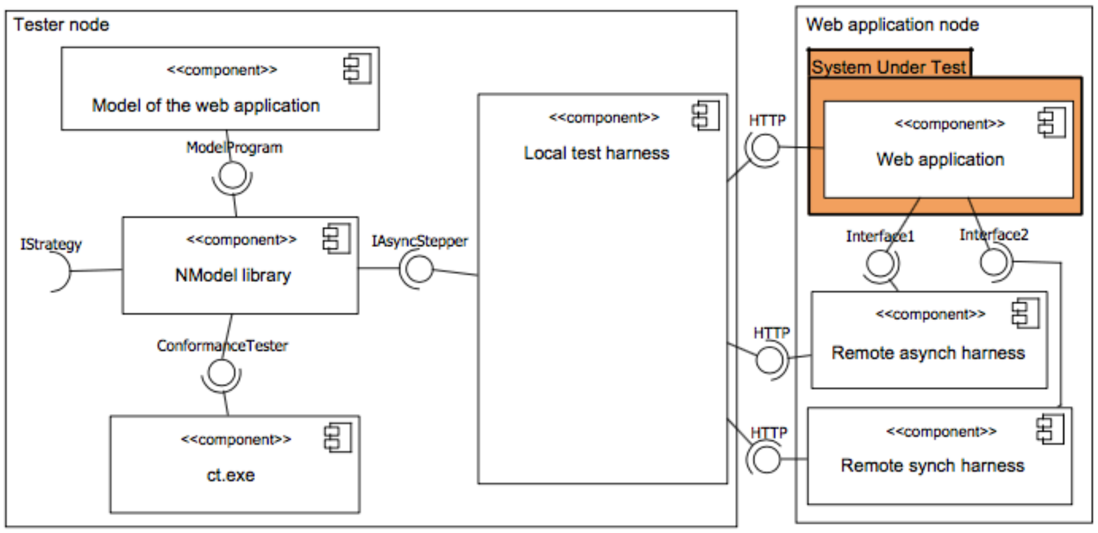


Figure 4 MBT setup for a web application [10].

The model of the SUT is available in the component labelled Model of the web application within the tester node.

The NModel Library is the central core that provides the tools for loading and stepping into the model program according to the desired functionalities.

Conformance testing was done by the conformance Tester component which loads the model program and the test harness. A test harness is the glue (connection) between the model program and the actual web application that was being tested. The whole conformance procedure is triggered by the ct tool within NModel toolkit. The test harness is exposed and called through IAsyncStepper (Asynchronous stepper) in NModel.

IAsyncStepper links synchronous actions within the model program to actual actions available in the implementation of the SUT. The case study, the actual synchronous (blocking) action corresponds to sending an HTTP request e.g GET, POST and waiting for the result to be delivered (response).

Functional specifications are derived in terms of causal relationships between different messages on from different ports on the system.
The SUT work such that the web server talks to the the backend system, which in turn communicates with the billing and positioning system. All procedure logs are transported to the history subsystem.

The following functionalities are modelled Login, LogOff, Positioning, BillingAndHistory and Restart as features in NModel.

2.3 MobiGUITAR Project: Automated MBT of Mobile Apps

MobiGUITAR automates GUI-driven testing of Android apps, employing an abstraction of GUI widgets' run-time state. The abstraction is a lable state machine model that, together with test coverage criteria, provides a way to automatically generate test cases [18].

The working of MobiGUITAR (Mobile GUI Testing Framework) involves an implementation of a toolchain that is android powered. The framework models the state of the app's (SUT) GUI.

There are three key steps to the working of MobiGUITAR including, ripping, generation and execution.

Ripping involves dynamically traversing of an app's GUI and creates a state machine model from this. MobiGUITAR ripper is an advanced version of the Android ripper, it launches the app in each start state and collects a list of events that can be performed on the GUI in the current state, with this a task list is created with the events as a separate task on the list. The Ripper traverse the task list which results into an event being fired according to the list, when

an event is fired, the event is removed from the list, new state occurs, the GUI changes and fireable events are collected and appended to the end of the list again.
This approach realizes a breadth-first traversal of the application's GUI in generating the state models. Base on some given criterion similar GUI are merged in a situation when the ripper sees them as equivalent using object properties constituting this GUI's to determine their equivalence. Equivalent states comprise equivalent objects whose IDs and type properties have the same values [18].

The steps explained above are used in testing a dictionary and offline reader called Aard Dictionary.

The process of test case generation involves using the model from the ripping process and test adequacy criteria to obtain test cases modelled as a sequence of GUI events. MobiGUITAR operates on a pairwise edge coverage criterion, to this end all pairs of adjacent events must be exercised together because of this all edges(events) adjacent to a node are created in pairs.

A pairwise edge coverage criterion was developed. Conceptually, this means that all pairs of adjacent edges (events) must be exercised together. To this end, we create pairs of all edges in our state machine that are adjacent to a node. For each pair, we generate a test case that's a path in the state machine from the start state to the pair being covered. For example, a2 has four incoming edges (el, ell, e12, and e13) and six outgoing edges (e7, e8, e9, e10, ell, and e12). This creates 4×6=24 pairs to cover. The test case (el, ell, e8) covers two of these pairs: (el, ell) and (ell, e8). For Aard Dictionary, we generated 678 test cases with 2,747 fireable events [18].

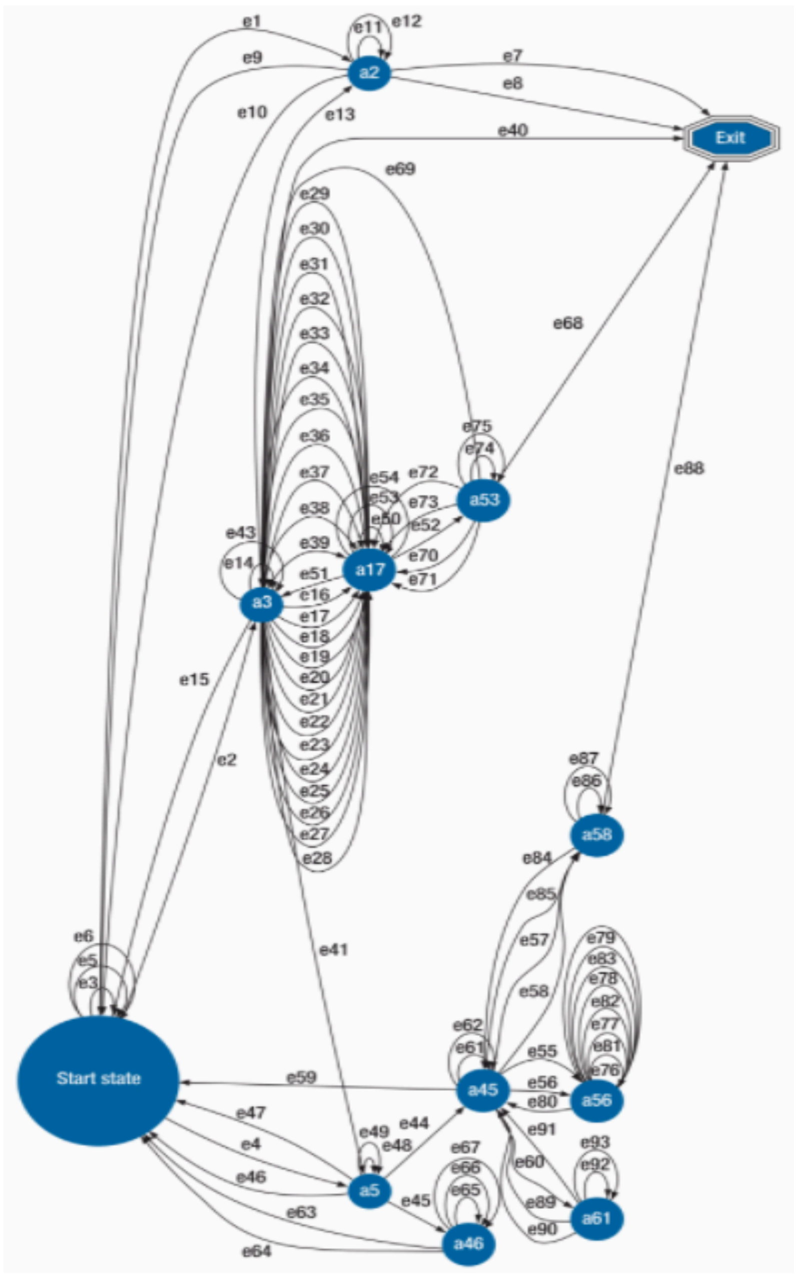


Figure 5 The abstract state machine for Aard dictionary and offline Wikipedia reader [18].

In the execution step of MobiGUITAR tests outputs from the generator which are in JUnit format are been replayed.
Such test cases can detect crashes during the app's execution. A tester can enhance a test case by adding JUnit-like assert statements to check for functional errors. For Aard Dictionary, 674 of the 678 test cases executed with no problems and covered 70 percent of the code. Of the 678 test cases, four detected a bug that led to an unhandled IllegalArgumentException [18].

2.4 Automated model-based Android GUI testing with multi-level GUI comparison criteria

One of the most widely used methodologies of testing mobile application is via its graphical user interface. The main goal of this approach is to validate the application through its views since the application itself is GUI driven.

As a means of GUI test automation for Android apps, current studies that deal with automated test input generation for android apps can be classified into several approaches. The simplest of this is random input generation, in which random inputs and events are generated as inputs to explore the behavior of the AUT while checking for runtime errors [11].

This research is main goal was to solve the problem that come along the line of using the random testing tools. The common problem associated with using these tools include the redundant test cases which are generated due to the nature/level of randomness, and with this approach no specific trace or path could be specified for testing.
For effective model-based GUI testing, reflecting a behavior space of an AUT on a GUI model is the most crucial ability to generate effective test inputs [19].

The core component of a systematic input generation is the underlying GUI model [11].
A GUI graph G is a directed graph that rep- resents distinct GUI states, distinguished by a specific GUI comparison criterion (GUICC), as ScreenNodes, and represents transitions between GUI states, which are triggered by events, as EventEdges. Simply, a GUI graph G is defined as G=(S,E), where S is a set of ScreenNodes (S)={s1,s2…sn}) and E is a set of EventEdges (E)={e1,e2…en}) [11].

When a model based testing, tool generates or updates a GUI graph while exploring the behavior space, GUICC distinguishes the GUI of the current screen on Android device after a test input is executed [11]. GUI comparison with GUICC helps the testing tool on how to update the GUI graph.

The multi-level GUI comparison technique was designed based on a semi-automated investigation with 93 real-word Android commercial apps registered in Google Play [11]. The result of this investigation is as follows:

The research came up with the GUICC model as 5 levels criteria with 3 possible outputs according to the comparison result. A comparison output can be either of *N(New), S(Same) or T(Terminated)* [11]. The default comparison level is set to level 5.
The comparison levels include:

- Level l: Compare package names
- Level 2: Activity names comparison
- Level 3: Compare layout widgets
- Level 4: Compare executable widgets
- Level 5: Content comparison

The research test setup is as follows:
A test environment which comprises of a testing engine which contains an error checked, graph generator and test executor. During test execution graph generator generates GUI graphs which is used in exploring the behavioral space of the AUT using the multi-level comparison criteria.

The comparison begins at level where the package name of the currently executing screen is compared to that of the visited ScreenNodes. To efficiently explore a behavior space (i.e., explore only the relevant space), a testing engine must be able to clearly distinguish the boundary of the space of an AUT. To avoid exploration outside of the app boundary, the testing engine analyzes which app package is currently being focused and run by the Android device. If the device does not focus on the package of the target AUT, the status of the target app is considered as a terminated state T (i.e., an exit state). This exit case can be caused by transitions to a third-party app, shutdown, or crashes by an executed event [11].

At level 2 after the comparison at level 1 is passed, the current activity name is compared in the same fashion as the package name in level 1. If not all ScreenNodes in the CUI graph have the same activity name, the current screen is considered as a newly discovered CUI state. On the contrary, if one or more ScreenNodes that have the same activity name are discovered in the CUI graph, the next level of comparison level 3 is taken [11].

At level 3 and 4 widget composition are the major criteria which are used for comparison. GUI layouts are compared in succession.

Level 5 is the last level of comparison, which is performed on a GUI which can be distinguished by its contents. In the case that multiple screens have separated context even if there is no difference in their widget compositions of two screens, modeling GUIs for each screen separately is necessary to represent behaviors that can act differently in different contexts [11].

The related works that have been describe leaves gap which is the fact that none of this random testing frame works is not considering observable features or behavior of their AUT. More generally, another opportunity for future research may lie in approaches for combining dynamic information obtained from system execution with information contained in the models [20].

3 SUT Modelling

All model programs in this thesis will be written in F# (.NET) using the NModel framework. NModel model programs uses the industry standard language c#. Interestingly a previous work by Anmol Gautam [21], has made it possible to be able to write model programs in F# which this thesis will put into good use.

Before diving into the modelling of our case study, it will be beneficial to understand some key concepts and features of NModel.

NModel supports modelling with abstract data structures including sets, maps as well as objects. Model programs can be composed of each other in a situation where features are splitted up into separate models.

NModel comes bundled with the following tools:

mpv tool which is used to visualize models and ensure that they behave as intended. Also design errors can be spotted in doing this.

otg offline test generator is used to generate test suite which contain test cases in the offline mode. test suite can later be passed to the *ct* to execute them.

ct is a test runner which executes by using a test harness for coupling an implementation. For this thesis and f# test harness will be written. The ct can either take a test suite generated by an ***otg*** or generated tests by itself on the fly. Using the model program supplied to it, with or without the option of a feature.

3.1 NModel Action

After completing the choice of the SUT behaviors which are to be modelled. this behavior or SUT requirements which are to be checked for conformance are wrapped within the action blocks of the model program.

In NModel the abstracted method which represents the feature or behavior of the SUT is decorated with `[Action]` symbol directly on the method. The enabling conditions which permits the operation of the MBT from one state to the other are determined by the value of the actions methods.

The action methods have zero or more arguments. the method is only enabled if the enabling condition evaluates to true.

3.2 NModel State Variables

When the model program is being run, different environmental variables are established which represents certain conditions in the internal of the SUT. The progress of the test, while it moves from one action to the other effects these variables causing them to take on new values from time to time as the test progresses.

3.3 Model Composition

With NModel SUT behaviors can be modelled in pieces as features which stands for a functionality or requirement of the system, NModel facilitates composition such that individual features modelled separately can be combined as during conformance testing.

3.4 Model Programs

A pre-condition to modelling our SUT is declaring those features of the SUT which are to be tested. We must never model a large system in one model instead we should make many small model programs [25].

Table 1 below is a list of requirements of our SUT which is the case study of this thesis. These are requirements crucial to the usage of the Moodle educational application on mobile.

Req1	A registered user should be able to access Moodle site from mobile
Req2	A user will be able to search a list of courses
Req3	A user will be able to do self-enrollment
Req4	Download and view some course resources
Req5	An enrolled user to a course will be able to view courses at a glance
Req6	An enrolled user to a course will be able to access the content of the course
Req7	A user will be able to access private messages, through the messages link

Table 1 - Test requirement list

The requirements above have been abstracted into smaller dedicated features.

A feature is a cluster of related state variables and actions that can be selectively included or excluded from a model program. Features make it possible to define different configurations for a model program, which include different subsets of features, in a single source program file (or collection of files) [1].

The combination of different features can be used in guiding a scenario which can translate to a requirement.

- Login
- Course self-enrollment
- Course search / Course Overview
- Course work submission

Appendix 1 shows the model program for the login feature from which the true FSM below is generated.

An FSM whose purpose is to define a scenario us called a *scenario FSM.* It is possible to define different scenario FSMs for any model program; each different scenario FSM defines a different collection of runs [1]. The FSM which can generate all the runs of a model program is called its *true* FSM [1].

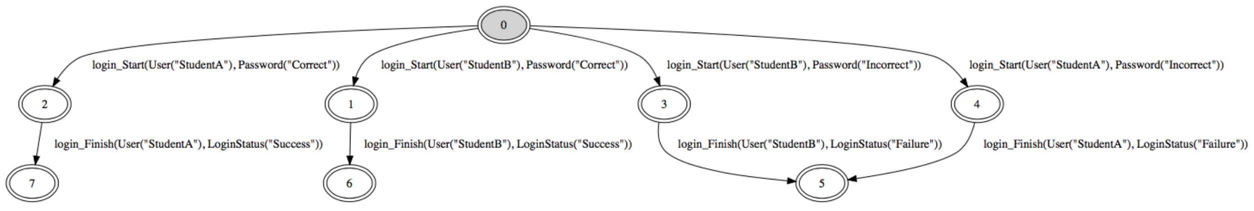

0
login_Start(User("StudentA"), Password("Correct"))
login_Start(User("StudentB"), Password("Correct"))
login_Start(User("StudentB"), Password("Incorrect"))
login_Start(User("StudentA"), Password("Incorrect"))
2
1
3
4
login_Finish(User("StudentA"), LoginStatus("Success"))
login_Finish(User("StudentB"), LoginStatus("Success"))
login_Finish(User("StudentB"), LoginStatus("Failure"))
login_Finish(User("StudentA"), LoginStatus("Failure"))
7
6
5

Figure 6 - Login Feature true FSM.

the FSM below shows a two user scenario where a login can be performed with either a correct or incorrect password or login credential. The left branch of the FSM as shown above is the scenario where both user's (Student A and Student B) credentials attempted login with a correct password which are shown on Node 0,2,7 and 0,1,6 respectively for both users. The right-side branch of the FSM is the scenario where login was attempted with incorrect password and login results to a Failure as shown on node 5.

A strength of NModel is in the fact that we can drill this down by guiding the model with a scenario which is supplied to the NModel as an FSM when generating the model FSM. Automated test input generation techniques attempt to generate a set of input values for a program or program component, typically with the aim of achieving some coverage goal or reaching a state (e.g., the failing of an assertion) [24].

In other to describe this we can demonstrate a login scenario with a single student, say Student A and a correct password. To this we can apply the following FSM text into the model generation command line argument:

```
FSM(
        0,
        AcceptingStates( ),
        Transitions(t(0,login_Start(User("StudentA"),Password("Correct")),1)),
        Vocabulary("login_Start")
)
```

the result of this is Figure 7 below:

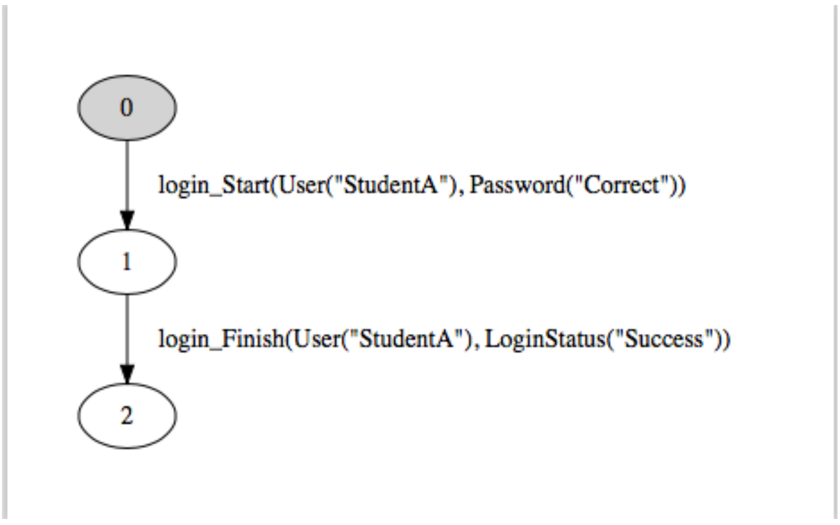

0
login_Start(User("StudentA"), Password("Correct"))
1
login_Finish(User("StudentA"), LoginStatus("Success"))
2

Figure 7 Single user with correct password, scenario FSM

A more interesting scenario to describe will be the course search scenario, where a logged in user is a requirement before a search can be made. There are several possible scenarios which can be generated from this action. Figure 8 below shows the gigantic FSM that is generated from the search feature model. Obviously, this is large and hard to interpret at a glance.

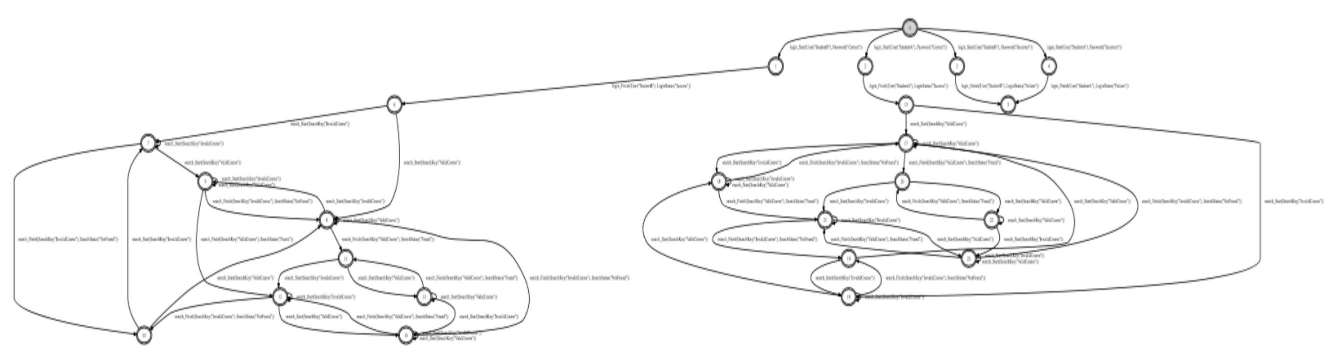


Figure 8 Search Feature true FSM

There are different scenarios in the true FSM above. For clarity, let use the FSM text below to narrow the true FSM down:

```
FSM(
        0,
        AcceptingStates(
                2
        ),
        Transitions(
                t(0,login_Start(User("StudentA"),Password("Correct")),1),
                t(1,search_Start(_),2)
        ),
        Vocabulary(
                "login_Start", "search_Start"
        )
)
```

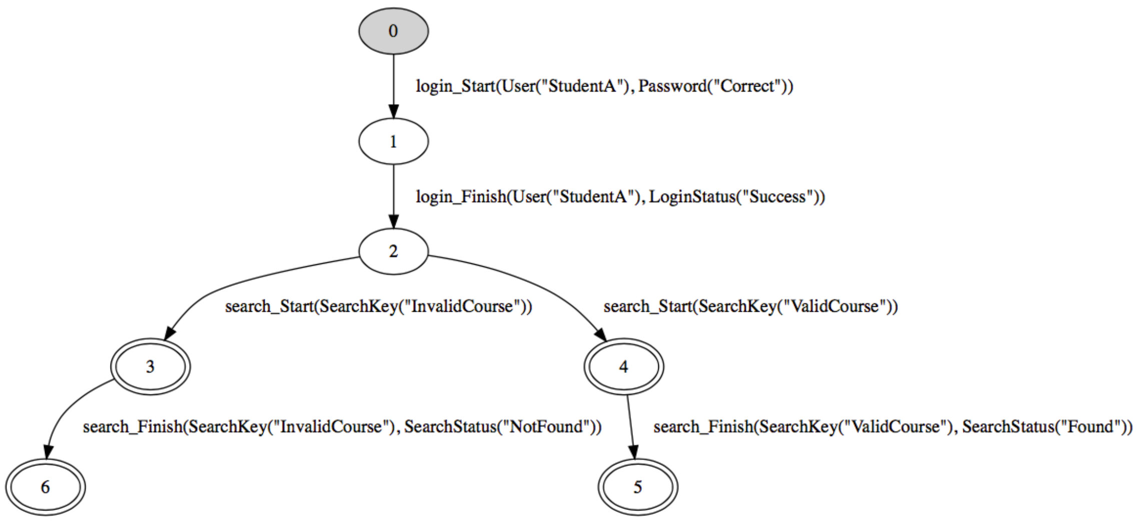

0
login_Start(User("StudentA"), Password("Correct"))
1
login_Finish(User("StudentA"), LoginStatus("Success"))
2
search_Start(SearchKey("InvalidCourse"))
search_Start(SearchKey("ValidCourse"))
3
4
search_Finish(SearchKey("InvalidCourse"), SearchStatus("NotFound"))
search_Finish(SearchKey("ValidCourse"), SearchStatus("Found"))
6
5

Figure 9 Single user search scenario FSM

The scenario FSM above is clear and it describes the behavior of automation when the conformance testing is performed on the model using a single student with a correct login credential and a valid course code is supplied as an argument during search. The advantage of this approach is that a tester can test and monitor the behavior of the AUT under any circumstance as the conformance tester generate several runs as shown in the large true FSM in Figure 8 above.

The other possible inputs which are generated to the various nodes during the generation of the model program as shown in the true FSM of our model in Figure8 is derived from the following enumerations:

```
type User = StudentA = 0 | StudentB = 1
type Password = Correct = 0 | Incorrect = 1
type LoginStatus = Success = 0 | Failure = 1
type SearchKey = ValidCourse = 0 | InvalidCourse = 1
```

From the enumeration values listed above, the test generator can generate a different combination that is passed to the action nodes of model program according to the parameter this action is expecting. To see how this is implemented reference can be made to both Appendix1 and Appenix2.

Let’s go further to describe other features. Below is the course self-enrollment feature which involves the two other features *Login* and *Search*, in the sense that a user must be logged in to perform the self-enrollment step and to the get view that shows the self-enrolment button, a course search operation must be performed, provided that the user is not already registered to this course.

Self-enrolment feature model program can be found in Appendix 3. The scenario FSM in Figure 10 below shows a scenario where student A is the currently logged in user and other possible scenarios where the course searched can either be a valid or invalid course, and the enrolment fulfillment can result to either of *Pending, Failed, Enrolled or Successful.*

The feature scenario is guided with the FSM text below.

```
FSM(
        0,
        AcceptingStates(
                3
        ),
        Transitions(
                t(0,login_Start(User("StudentA"),Password("Correct")),1),
                t(1,search_Start( _ ),2),
                t(2,enrol_Finish( _ , _ ),3)
        ),
        Vocabulary(
                "login_Start", "search_Start", "enrol_Finish"
        )
)
```

In the FSM text above the underscore symbol '*_*' passed as argument to the *search_Start* and *enroll_Finish* actions are so that the model viewer tool can generate all possible arguments to these actions respectively.

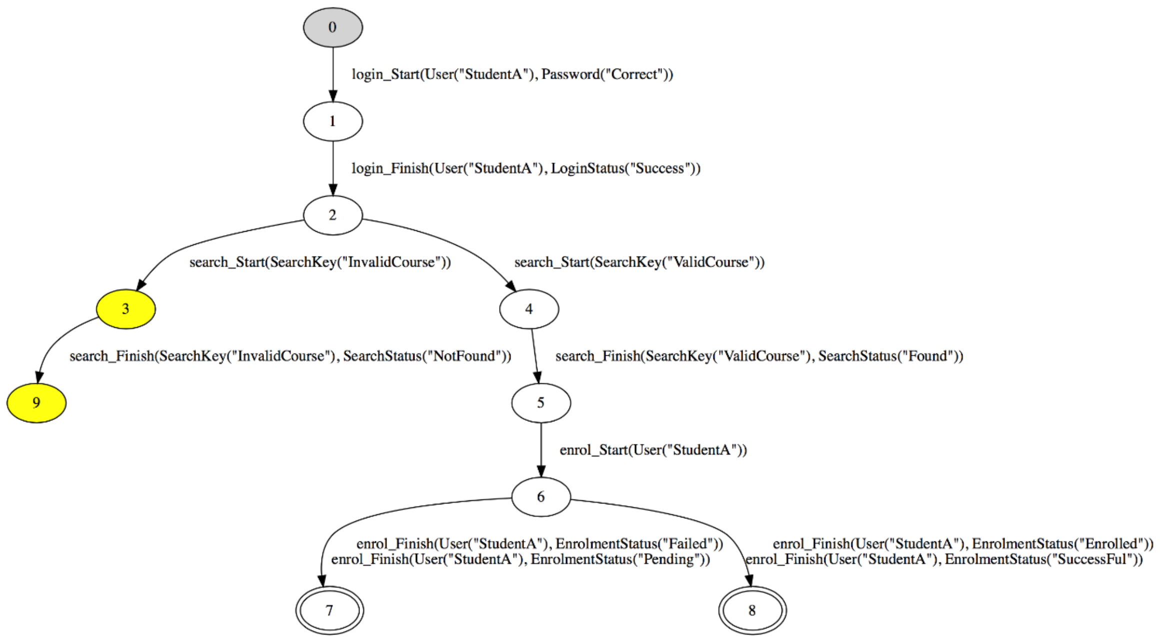

0
login_Start(User("StudentA"), Password("Correct"))
1
login_Finish(User("StudentA"), LoginStatus("Success"))
2
search_Start(SearchKey("InvalidCourse"))
search_Start(SearchKey("ValidCourse"))
3
4
search_Finish(SearchKey("InvalidCourse"), SearchStatus("NotFound"))
search_Finish(SearchKey("ValidCourse"), SearchStatus("Found"))
9
5
enrol_Start(User("StudentA"))
6
enrol_Finish(User("StudentA"), EnrolmentStatus("Failed"))
enrol_Finish(User("StudentA"), EnrolmentStatus("Pending"))
enrol_Finish(User("StudentA"), EnrolmentStatus("Enrolled"))
enrol_Finish(User("StudentA"), EnrolmentStatus("SuccessFul"))
7
8

Figure 10 Scenario FSM for self-enrolment feature

State 3 and 9 are dead states as marked in the Figure 10 above, while state 7 and 8 are the accepting state in which the test can be marked as completed. Adding the liveness parameter to model generator command we can view the dead states in the model program which are filled in color yellow above.

Course work submission model like the self-enrolment is a feature that combines all of the other features that have been previously described. i.e Authentication is a pre-condition so the Login feature is required to complete the course work submission task.
A user needs to login into the app, search for the course, from the course overview page, course activities can be viewed from which a selection can be made, and then the user can add a submission to the course activity that was previously selected. This is visualized in Figure 11 below we can see clearly the accepting states which marked in nodes 10 for when the submission is success and node 11 for when the submission fails.

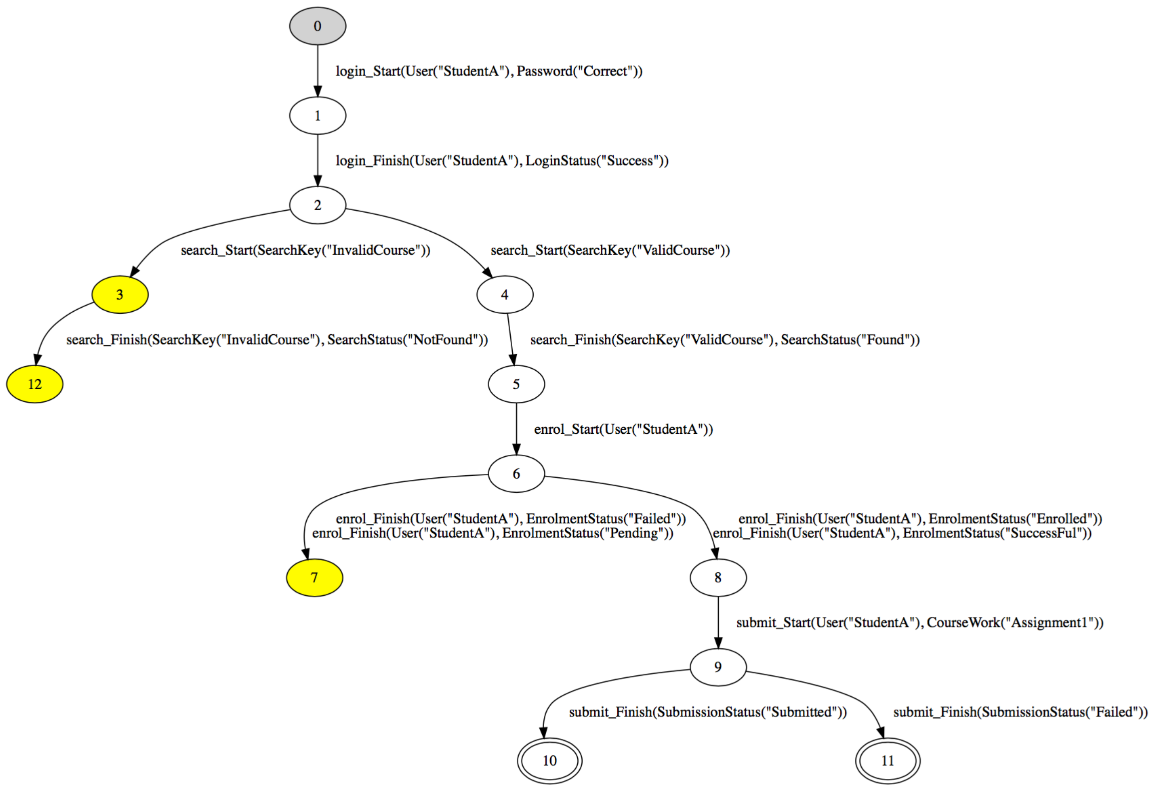

0
login_Start(User("StudentA"), Password("Correct"))
1
login_Finish(User("StudentA"), LoginStatus("Success"))
2
search_Start(SearchKey("InvalidCourse"))
search_Start(SearchKey("ValidCourse"))
3
4
search_Finish(SearchKey("InvalidCourse"), SearchStatus("NotFound"))
search_Finish(SearchKey("ValidCourse"), SearchStatus("Found"))
12
5
enrol_Start(User("StudentA"))
6
enrol_Finish(User("StudentA"), EnrolmentStatus("Failed"))
enrol_Finish(User("StudentA"), EnrolmentStatus("Pending"))
enrol_Finish(User("StudentA"), EnrolmentStatus("Enrolled"))
enrol_Finish(User("StudentA"), EnrolmentStatus("SuccessFul"))
7
8
submit_Start(User("StudentA"), CourseWork("Assignment1"))
9
submit_Finish(SubmissionStatus("Submitted"))
submit_Finish(SubmissionStatus("Failed"))
10
11

Figure 11 Coursework submission scenario FSM

The transition from one state to the other is basically a result of action guards which are put in place, meaning that an action of the model program is not enabled not until the all the prerequisite conditions evaluates to true, and this is very important as we can use this in expressing the behavior of an application in different scenario.

Let us examine the sample code below which is the submission feature model program

```
namespace MoodleModel
open System
open NModel
open NModel.Attributes
open NModel.Execution
type CourseWork = Assignment1 = 0
type SubmissionStatus = Submitted = 0 | Failed = 1


[<Feature("Submit")>]
type MoodleSubmitApp() =
    static member val view = Screen.Submission with get, set
    static member show() = Console.WriteLine("Current View is" + MoodleSubmitApp.view.ToString())
    static member val submissionFinished = false with get, set
    static member val currentUser : Set<User>  = Set<User>.EmptySet with get,set
    static member val submissionQueue : Set<CourseWork>  = Set<CourseWork>.EmptySet with get,set


    static member submit_StartEnabled(student: User, submission : CourseWork) =
         MoodleEnrolApp.courseParticipants.Contains(Pair<User,EnrolmentStatus>(student,EnrolmentStatus.SuccessFul))
          && MoodleSubmitApp.submissionQueue.Count < 1 && MoodleSubmitApp.submissionFinished = false
    [<Action>]
    static member submit_Start(student: User, submission) =
        MoodleSubmitApp.show()
        MoodleSubmitApp.submissionQueue <- MoodleSubmitApp.submissionQueue.Add(submission)

    static member submit_FinishEnabled(taskStatus: SubmissionStatus) = MoodleSubmitApp.submissionFinished = false
        && MoodleSubmitApp.submissionQueue.Count > 0
    [<Action>]
    static member submit_Finish(taskStatus)=
        MoodleEnrolApp.show()
        MoodleSubmitApp.submissionQueue <-
        match taskStatus with
        | SubmissionStatus.Submitted -
> MoodleEnrolApp.view <- Screen.CourseOverview ; MoodleSubmitApp.submissionFinished<- true ; Set<CourseWork>.EmptySet
        | _ -> MoodleSubmitApp.submissionQueue
```

The action guard to enable a submission as shown in the snippet above is for the

i. course participant’s collection(Map) to contain a key with the currently logged in user (student) and the corresponding value which is the enrollment status to be Successful,

ii. The length of the submission queue must be empty and

iii. Submission completion tracker *submissionFinished* must be false

These guards will ensure that the action they are tied to can only be executed when all these conditions are evaluated to true. This pattern also applied to the completion face which checks whether there is a submission in progress, by checking the length of the *submissionQueue* and ensuring that the submission have not been previously finished.

4 Test Setup

4.1 Required applications and tools

In this chapter, the setup that is used for the test in this research will be discussed. The following software/tools are used in this thesis.

a. Moodle Software (Mobile and Desktop).

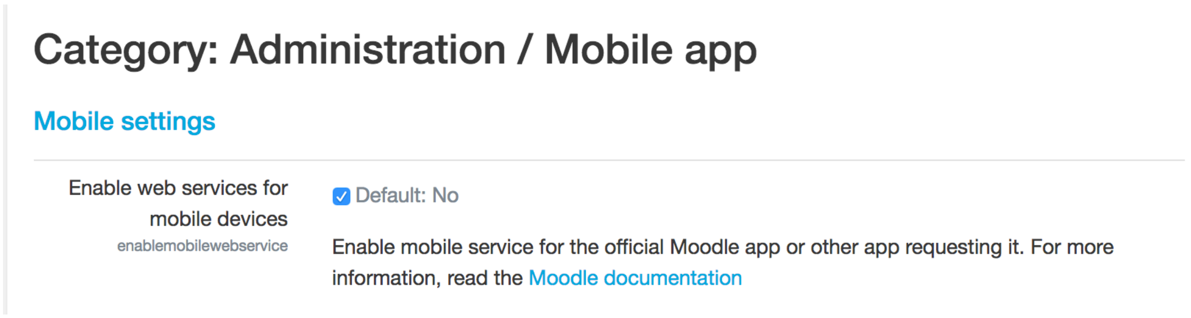
b. Appium

c. Apache server

4.1.1 Moodle Software

Moodle educational software is the popular educational software, it is a learning platform that is developed for personalized usage.

Moodle mobile application is the SUT, this is installed on a android device which is connected to the desktop version of the software in motion. To achieve this, the web service option of the desktop application (Moodle) needs to enable, which is by default set to `No`, Figure 4 below shows the site administration page of Moodle desktop software where this option can be turned on.



Category: Administration / Mobile app

Mobile settings

Enable web services for mobile devices
enablemobilewebservice

✓ Default: No

Enable mobile service for the official Moodle app or other app requesting it. For more information, read the Moodle documentation

Figure 12 Enabling mobile web service in moodle site administration menu

Moodle web application is needed to create courses and other resource related to our tests setup. Moodle mobile is a hybrid mobile application. A hybrid application is an application developed with HTML, JavaScript and CSS and then wrapped into a native wrapper for the target mobile platform which can either be Android or IOS.

Moodle mobile application was installed on an android device (Huawei LYO-L21) running Android OS v5.1. To carry out tests on the android device USB debugging must be toggled on the device through the developer options menu on the device.

4.1.2 Appium

Appium is an open source tool used in automating native, mobile web and hybrid applications on both Android and IOS platform. With Appium multiple tests can be written targeting different platforms with same API. Appium's automation is as in Selenium where all the application containing elements are exposed and can be accessed using permitted operations like clicks, taps, shakes, swiping and send keys to input fields.

The choice of Appium over other alternatives; it is open source and has an active community of developers. The Appium framework is used as this work intend to provide the technology to be used by the Telia Testlab.

The key concept behind the working of Appium is the client/server architecture, Appium server is a Nodejs server, which listens to regular http requests (POST/GET) and then transforms the request using the internal JSON wire protocol the output of this is passed to the instrumentation layer of the device which has been implemented in the core of Appium

A driver is required to perform automations on devices. An Appium driver can be written in any of the following languages, C#, JavaScript, Java, Python.
This thesis an F# Appium driver was written with the same official bindings for the C# driver.

The driver is connected to the IStepper interface of the NModel program, which all our model programs will execute when actioned with the conformance testing tool.

The target device must be selected on the Appium client, this can either be an emulator or a real device, our case we selected out device on which we have installed the Moodle application.

4.1.3 Apache Web Server

Apache is the mostly widely used web server software [26]. A web server is a software that accepts HTTP requests and respond to such requests accordingly.

Apache was installed on a cloud server, which is where the Moodle desktop software is deployed. A cloud server is not compulsory, the desktop software can be installed on a local machine. The choice of the cloud server was to test the Moodle web service on a remote system, which is the way this we be in production setup.

The tools described above are chained as follows, as shown in the diagram below:

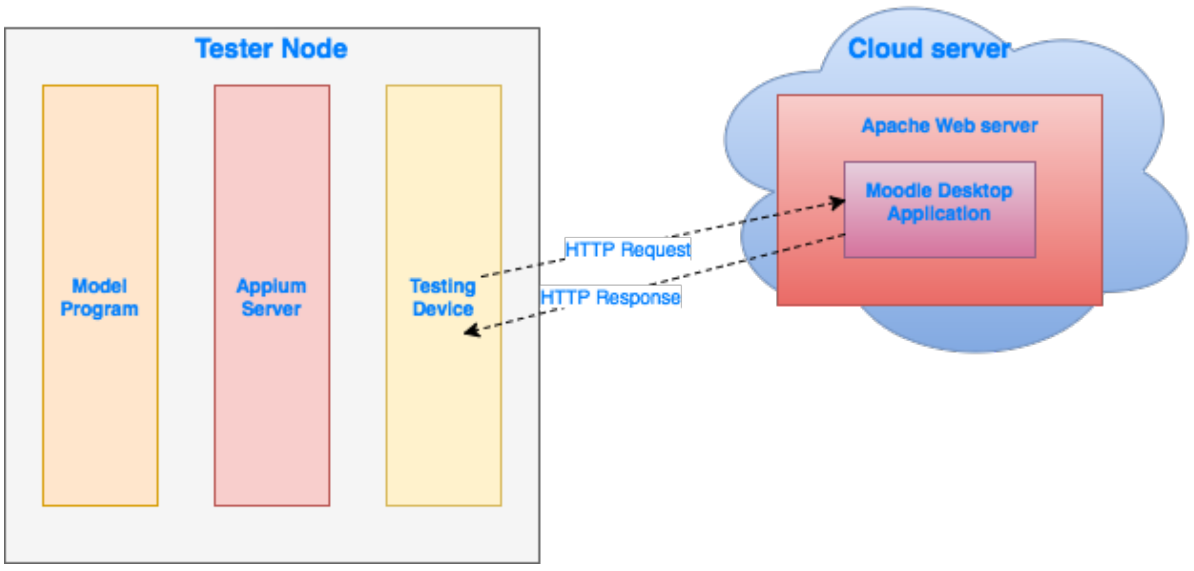


Figure 13 Test Setup

Figure 13 above shows the overall architecture of our setup. The tester node comprises of our model programs, Appium server and our testing device. The cloud server maintains the Apache server, the Moodle desktop application is served.

The interface between the tester node and the cloud server is HTTP web service request, though which various requests from the mobile application are sent.

The interface between the model program and the Appium server is through the system adapter described in the next section.

5. System Adapter

The stepper is the binding interface between the different AUT behaviors modelled in our model programs and the actual application execution. This is the pipeline through which the various action triggered by the conformance tester *ct* tool of NModel gets to the actual application which sits on the mobile device.

The adapter is developed using the Appium Framework, and the implementation is done using F#. The adapter contains different methods which may be directly translated to an equivalent action in the model program, depending on the implementation approach.

Appendix 4 shows the source code of the stepper, which is an implementation for the IStepper interface which get the actions symbols of the model program during execution, with the action symbol the stepper decides the equivalent action to be taken by the automation driver. The automation driver is an Appium f# driver implementation class in Appendix 5 with methods which contains series of steps executed within the application, a step can be an input entry, button click, a look for a view element.

The Appium framework exposes almost equivalent methods exposed by the selenium web driver.

The entire process of achieving MBT testing of a mobile application starting from the selection and modelling of AUT requirements stage to the implementation of an actual stepper that automates the actions modelled can be described in the following steps

Step 1: Base on the requirements we have in Table 1 above, a tester will select some the behavior of the application to be tested, and the recommendation in this case to group behaviors into features, which will result in to having small model programs which are concise and enable easy abstraction of the MBT behavior.
A sample feature modelled is the Login feature in Appendix 1.

Step 2: After the modelling task has been completed, the next thing is for us to visualize and make sense of the what we have modelled. An FSM can be generated using the *mpv* tool, which comes bundled with NModel.

This is achieved by running the *mpv* command on the terminal:

```
mpv /r:MoodelModel.dll MoodleModel.Factory.Create
```

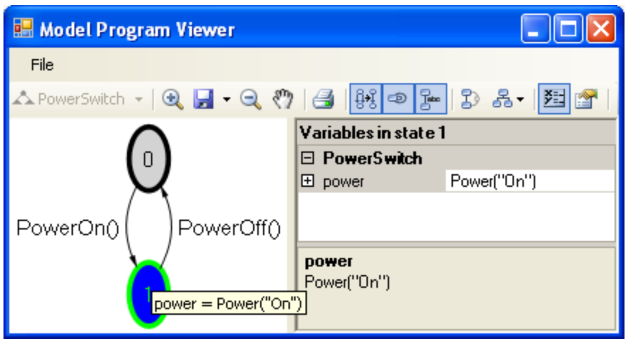


Figure 14 Model program viewer window [24] .

Figure 12 above shows the mpv display windowon Windows operating system. The mpv tool performs exploration, which generates a finite state machine (FSM) from a (possibly "infinite") model program. The tool displays the graph of the generated FSM [24].

As of the time of this work, the only option available for the MAC operating system is to follow the process of generating a dot file from which a pdf is generated for viewing the model program.

```
mono ~/nmodelTools/binaries/mp2dot.exe
/mp:MoodleModel[Login,Search,Enrol,Submit]
/nodeLabelsVisible+
/r:../MoodleModel.dll
/dotFileName:submit_studentA.dot /fsm:submit_SubmitA.fsm
/livenessCheckIsOn+
```

Running the command above from the terminal generates a dot file. Different options can be passed to the to the mp2dot.exe command, for a full usage, execute

```
mono ~/nmodelTools/binaries/mp2dot.exe /?
```

To convert this dot file to a pdf file that can be viewed we need to run the dot command specifying the output format and the input dot file name as follows:

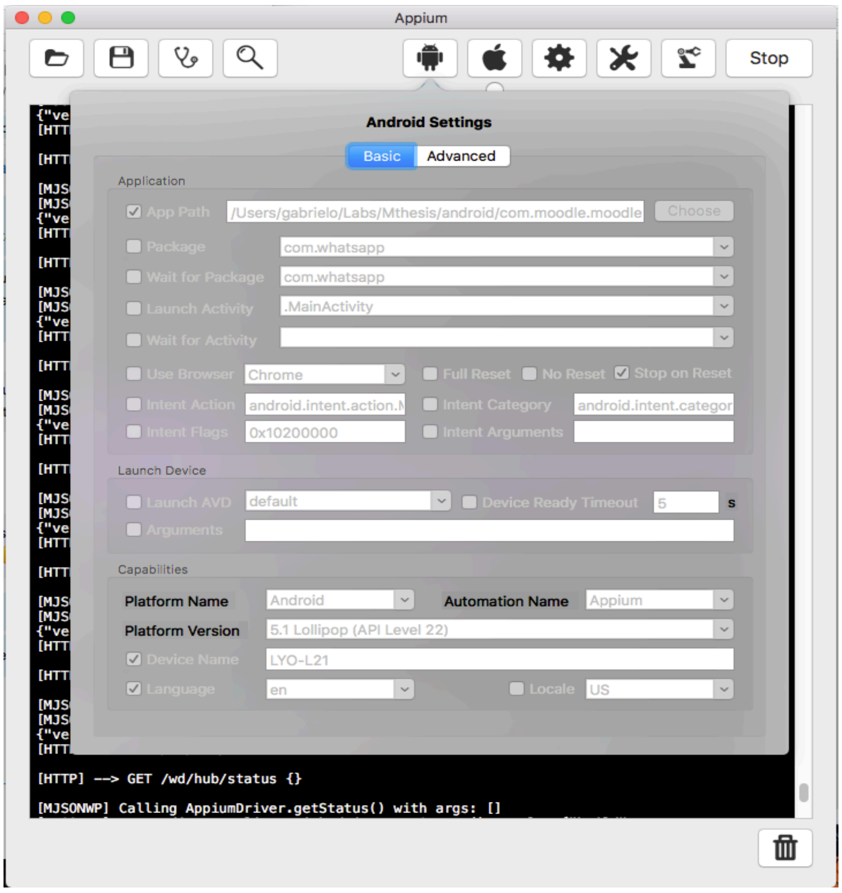
```
dot -Tpdf submit_studentA.dot -o submit_studentA.pdf
&& open submit_studentA.pdf
```

the first parameter is the output format we are targeting, there are other formats available with the dot library, in our case we want a pdf file so we specify `–Tpdf` and the second parameter is the input dot file name `submit_studentA.dot` the next parameter is the output file name which is specified with a flag `-o submit_studentA.pdf`

The result of this is shown in Figure 11 above.

This process can be repeated as required. If all is well and the tester is comfortable with the model the next step is to prepare for automation.

Step3: this step involves the setting up the needed tools/frameworks upon which our automation driver depends. We need install Appium client and server, which is available for both windows and mac operating systems. Installation guide for Appium is available on the project website here; http://appium.io/



Application
- [✓] App Path: /Users/gabrielo/Labs/Mthesis/android/com.moodle.moodle
- [ ] Package: com.whatsapp
- [ ] Wait for Package: com.whatsapp
- [ ] Launch Activity: .MainActivity
- [ ] Wait for Activity:
- [ ] Use Browser: Chrome
- [ ] Full Reset
- [ ] No Reset
- [✓] Stop on Reset
- [ ] Intent Action: android.intent.action.M
- [ ] Intent Category: android.intent.categor
- [ ] Intent Flags: 0x10200000
- [ ] Intent Arguments:

Launch Device
- [ ] Launch AVD: default
- [ ] Device Ready Timeout: 5 s
- [ ] Arguments:

Capabilities
- Platform Name: Android
- Automation Name: Appium
- Platform Version: 5.1 Lollipop (API Level 22)
- [✓] Device Name: LYO-L21
- [✓] Language: en
- [ ] Locale: US

```
[HTTP] --> GET /wd/hub/status {}
[MJSONWP] Calling AppiumDriver.getStatus() with args: []
```

Figure 15 Appium Client Window on Mac

Figure 13 above shows the Appium client on mac. The client application we can setup the target device as well as the operating system. On the client application, we can inspect application view elements properties including *ids, style, xpath* e.t.c

In this work, the xpath strategy was used because, the only unique attribute with which view elements in the Moodle application which is a Cordova application can be accessed is the xpath. This was a challenge during this work. In general, other strategies presented by the Appium driver class can be used, this which is a function of the application in question.

Step4: On a successful setup of the Appium client and server which is the core component required by the stepper class, we can begin to implement our stepper as shown in Appendix 6.

Step5: This is the conformance testing phase where we validate our model program(s) against the AUT using an emulator or an actual device. This is achieved by running the *ct* command from a terminal window after starting the Appium server.
Execute command below in a terminal window,

```
mono ~/nmodelTools/binaries/ct.exe /r:MoodleStepper.dll
/iut:AppStepper.Stepper.Create
/r:../../../../MoodleModel/MoodleModel/bin/Debug/MoodleModel.dll
/mp:MoodleModel[Login] /timeout:20000
/fsm:../../../../MoodleModel/MoodleModel/bin/Debug/scenarios/onecorrectuser
scenario.fsm
```

The command above is readable except for the additional options passed to the command, this can vary as required. For the full usage of the *ct* tool, execute the command below in the terminal.

```
mono ~/nmodelTools/binaries/ct.exe /?
```

On executing the conformance test, the test reports are reported can be seen on the console or in text file if one is specified to the ct command, the test output will be written to this file.

5.1 Test Results and Observations

All tests are carried out on a real device. A summary of the tests conducted and the outcomes are inline in Table 2 below.

	Test Description	Expected	Result
1.	Connect a site desktop site	Login page should be displayed	Passed
2.	Login with incorrect password	Login should fail with adequate error message displayed.	Passed
3.	Login with correct password	Dashboard screen should be displayed	Passed
4.	Search for an available course	The search screen should return a list of containing the searched course	Passed
5.	Search for an unavailable course	An empty list should be returned	Passed
6.	Enroll for a course	Course enrolment should be successful and course should be listed on the dashboard	Passed
7.	Check course overview	Details of the course should be displayed with related course activities listed	Passed
8.	Submit course work	Course work should be successfully submitted using the text area input of the course work submission screen	Passed

Table 2 Summary of test results

All test is conducted combining separate features together depending on the requirement features. For instance, the course work submission test requires combining [Login, Search, Submit]

Tests are conducted using applicable scenario FSM to guide the test to have the specific test required. The tests are carried out on mac OS system using mono which shows that this setup is fine with mac OS.

Overall we have successfully chained all tools and achieved MBT testing in a behavioral pattern using NModel and Appium

Conclusion and Recommendation

This thesis has presented a behavioral approach to testing Android mobile applications using NModel, Appium and Moodle. A system adapter that binds model programs written in F# to actual application execution via JSON wire protocol has also been implemented.

Behavioral approach to testing mobile applications using models and strategic scenarios can be used in testing complex applications with different dynamics. Using this approach for testing mobile applications can affect the way mobile applications are developed.

One key challenge experienced during this work was the visual element selection using the Appium client application inspector, with this tool, the only attribute that distinguished visual elements is its xpath in the Moodle mobile application, and because of this, there are long parameter values supplied to the selector in form of xpath strings as shown in Appendix 6, normally it should be possible to target visual elements using other strategies, but the attributes to this strategies are not available in the Moodle application.

Future works that can be done to improve MBT with NModel can be in form of building tools that replaces terminal operations, as this seem to be a difficult operation for an average tester with minimal terminal experience. Behavioral test framework integration e.g Tickspec, with this, models can be combine with FSM's for test, this will make test readable and understandable.

References

[1] J. Jacky, M. Veanes, C. Campbell and W. Schulte, Model-Based Software Testing and Analysis with c#.

[2] S. Admin, "QATestingTools.com," 27 December 2015. [Online]. Available: http://www.qatestingtools.com/testing-tool-article/model-based-testing.

[3] R. Hower, "What kind of testing should be considered," [Online]. Available: http://www.softwareqatest.com/qatfaq1.html#FAQ1_5.

[4] "Integration Testing," [Online]. Available: http://softwaretestingfundamentals.com/integration-testing/.

[5] "Definition of System Testing," [Online]. Available: http://economictimes.indiatimes.com/definition/system-testing.

[6] A. Abadi, D. Dig and E. Tillevich, "MobileDeli'14 Workshop: welcome message of the chairs".

[7] É. Cota, "Embedded software testing: what kind of problem is this?".

[8] Microsoft, "Model Based Testing," [Online]. Available: https://msdn.microsoft.com/en-us/library/ee620469.aspx.

[9] Christian, "Model-based testing. An introduction to benefits and drawbacks," [Online]. Available: http://testtech.dk/content/model-based-testing-introduction-benefits-and-drawbacks.

[10] J. Ernits, R. Roo, J. Jacky and M. Veanes, "Model-Based Testing of Web Applications using NModel.," 2009.

[11] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using Multi-level GUI comparison criteria.".

[12] appium.io, "Appium," [Online]. Available: http://appium.io/.

[13] S. T. Material, "Monkey Talk," [Online]. Available: https://softwaretestingbin.blogspot.com.ee/p/monkey-talk-monkey-talk-is-open-source.html.

[14] calaba.sh, "calaba.sh," [Online]. Available: calaba.sh.

[15] Robotium, "Robotium," [Online]. Available: https://robotium.com/.

[16] Selendroid, "Selendroid," [Online]. Available: http://selendroid.io/.

[17] N. D. Hoai, P. Strooper and J. G. Süß, "Automated functionality testing through GUIs," 2010.

[18] D. Amalfitano, A. R. Fasolino and P. Tramontana, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," 2014.

[19] G. N. K. S. W. Choi, "Guided gui testing of android apps with minimal restart and approximate learning," vol. 48, no. 10, pp. 623-640, Oct 2013..

[20] A. Orso and G. Rothermel, "Software Testing: A Research Travelogue (2000 – 2014)".

[21] A. Gautam, "NModel Fsharp: Modelling in F# for Model Based Testing.," 2016.

[22] S. J. A. U. Monalisa Sarma, "Model-Based Testing in Industry – A Case Study with Two MBT Tools.".

[23] J. Jacky, "PyModel: Model-based testing in Python," in *Proc. of The 10th Python in Science Conference. (SCIPY 2011)*.

[24] M. P. Viewer, "Codeplex.com," [Online]. Available: https://nmodel.codeplex.com/wikipage?title=Model%20Program%20Viewer&referringTitle=Home.. [Accessed May 2017].

Appendix 1 – Login Feature Model

```
namespace MoodleModel

open System
open NModel
open NModel.Attributes
open NModel.Execution

type Screen =Login = 0| Dashboard = 2 | CourseSearch = 3 | CourseOverview = 4 | Submission =5
type public User = StudentA = 0| StudentB = 1
type Password = Correct = 0| Incorrect = 1
type public LoginStatus = Success = 0 | Failure = 1

[<Feature("Login")>]
type MoodleApp() =
    static member val view = Screen.Login with get, set
    static member show() = Console.WriteLine("Login ::" + MoodleApp.view.ToString());

    static member val currentLogins = Map<User,LoginStatus>.EmptyMap with get, set
    static member val currentUser : Set<User>  = Set<User>.EmptySet with get,set

    static member login_StartEnabled() = MoodleApp.view = Screen.Login
    [<Action>]
    static member login_Start (user : User , password : Password)=
        MoodleApp.show()
        MoodleApp.view <- Screen.Dashboard
        if password = Password.Correct then
           MoodleApp.currentLogins <- MoodleApp.currentLogins.Add (user, LoginStatus.Success)
        else
           MoodleApp.currentLogins <- MoodleApp.currentLogins.Add (user, LoginStatus.Failure)

    static member login_FinishEnabled (user, loginStatus) =
        MoodleApp.currentLogins.Contains (Pair<User,LoginStatus> (user, loginStatus))
    [<Action>]
    static member login_Finish (user : User, loginStatus) =
        if loginStatus = LoginStatus.Success then
            MoodleApp.currentUser <- Set<User> (user)
        else
            MoodleApp.currentUser <- Set<User>.EmptySet
        MoodleApp.currentLogins <- MoodleApp.currentLogins.RemoveKey (user)


type Factory() =
    static member Create() =
        LibraryModelProgram (typedefof<Factory>.Assembly, "MoodleModel")
```

Appendix 2 – Search Feature Model

```
namespace MoodleModel

open System
open NModel
open NModel.Attributes
open NModel.Execution

type public SearchStatus = Found = 0 | NotFound = 1
type public SearchKey = ValidCourse = 0 | InvalidCourse = 1



[<Feature("Search")>]
type MoodleSearchApp() =
    static member val view = Screen.Dashboard with get, set
    static member show() = Console.WriteLine("Search :: " + MoodleApp.view.ToString());
    static member log(message : string) = Console.WriteLine(message);

    static member val foundCourse = Set<SearchKey>.EmptySet with get, set
    static member val currentSearch = Map<SearchKey,SearchStatus>.EmptyMap with get, set
    static member val searchStarted = false with get, set

    static member search_StartEnabled() =  MoodleApp.currentUser.Count > 0
    [<Action>]
    static member search_Start(keyWord : SearchKey) =
        MoodleSearchApp.log("Search Started");
        MoodleSearchApp.show()
        MoodleSearchApp.view <- Screen.CourseSearch

        if keyWord = SearchKey.ValidCourse then
            MoodleSearchApp.currentSearch <- MoodleSearchApp.currentSearch.Add (keyWord, Sear
chStatus.Found)
        else
            MoodleSearchApp.currentSearch <- MoodleSearchApp.currentSearch.Add (keyWord, Sear
chStatus.NotFound)

    static member search_FinishEnabled(searchKeyType:SearchKey,searchOutcome:SearchStatus) =
        MoodleSearchApp.currentSearch.Contains (Pair<SearchKey,SearchStatus> (searchKeyType,
searchOutcome))
    [<Action>]
    static member search_Finish(searchKeyType: SearchKey, searchOutcome : SearchStatus) =
        MoodleSearchApp.log("Search Finished")

        if searchOutcome = SearchStatus.Found then
            MoodleSearchApp.foundCourse <- Set<SearchKey>(searchKeyType)
        else
            MoodleSearchApp.foundCourse <- Set<SearchKey>.EmptySet
        MoodleSearchApp.currentSearch <- MoodleSearchApp.currentSearch.RemoveKey (searchKeyTy
pe)
```

Appendix 3 – Self-Enrolment Feature

```
namespace MoodleModel

open System
open NModel
open NModel.Attributes
open NModel.Execution


type public EnrolmentStatus = Pending=0 | Successful= 1 | Failed = 2 | Enrolled = 3


[<Feature("Enrol")>]
type MoodleEnrolApp() =
    static member val view = Screen.Dashboard with get, set
    static member show() = Console.WriteLine("Enrol :: " + MoodleApp.view.ToString());
    static member log(message : string) = Console.WriteLine(message);
    static member val enrolStarted = false with get, set
    static member val courseParticipants = Map<User,EnrolmentStatus>.EmptyMap with get, set

    static member enrol_StartEnabled( student ) =  MoodleSearchApp.foundCourse.Count > 0 &&
        not (MoodleEnrolApp.courseParticipants.ContainsKey(student)) && MoodleApp.currentUser.
Contains( student)
    [<Action>]
    static member enrol_Start(student : User) =
        MoodleEnrolApp.log("Enrolment Inprogress");
        MoodleEnrolApp.view <- Screen.CourseSearch
        MoodleEnrolApp.show()
        MoodleEnrolApp.courseParticipants <- MoodleEnrolApp.courseParticipants.Add(student, E
nrolmentStatus.Pending)
        MoodleEnrolApp.enrolStarted <- true


    static member enrol_FinishEnabled(student:User, actionStatus:EnrolmentStatus) =
        (MoodleEnrolApp.enrolStarted = true) && (MoodleEnrolApp.courseParticipants.Contains(P
air<User,EnrolmentStatus>(student,EnrolmentStatus.Pending)))
    [<Action>]
    static member enrol_Finish(student, actionStatus) =
        MoodleEnrolApp.log("Enrolment Completed")
        if actionStatus = EnrolmentStatus.SuccessFul then
            MoodleEnrolApp.view <- Screen.CourseOverview
            MoodleEnrolApp.courseParticipants <- MoodleEnrolApp.courseParticipants.RemoveKey(
student)
            MoodleEnrolApp.courseParticipants <- MoodleEnrolApp.courseParticipants.Add(Pair<U
ser,EnrolmentStatus>(student, EnrolmentStatus.SuccessFul))
        else
            MoodleEnrolApp.view <- Screen.CourseSearch
        MoodleEnrolApp.enrolStarted <- false
```

Appendix 4 – Stepper Class

```
namespace AppStepper

open NModel.Conformance
open NModel.Terms
open System
open System.IO

open AppiumActions

type Stepper()  =
    let appiumCaller = new Actor()
    do appiumCaller.Init ()

    static member  Create() =
        new Stepper ()

    interface IStepper with
        member this.DoAction (a) =
            match a.FunctionSymbol.ToString() with
            //| "appLaunched" -> appiumCaller.connectSite ()
            | "login_Start" -> appiumCaller.logIn (a)
            | "courseSearch" -> appiumCaller.courseSearch ()
            | "courseEnrollment" -> appiumCaller.enrollForCourse()
            | "courseSubmission" -> appiumCaller.addSubmission()

        member this.Reset() =
            Actor.driver.Quit()
            appiumCaller.Init ()
```

Appendix 5 – F# Appium driver for Moodle App

```
namespace AppStepper

open NModel.Conformance
open NModel.Terms
open System
open System.IO

open OpenQA.Selenium
open OpenQA.Selenium.Appium
open OpenQA.Selenium.Remote
open OpenQA.Selenium.Appium.Android

open MoodleModel



module utils =

    let log (text : string) =
        System.Console.Error.WriteLine(text)

    let wait (duration:int) =
        log("Now waiting for " + string(duration) + " seconds")
        System.Threading.Thread.Sleep(duration)


open utils

module AppiumActions =
    type Credentials = { Username : String ; Password : String }

    type Actor() =
        let STUDENT_CREDENTIAL = Map.empty.Add("correct", { Username = "student" ; Password =
 "c#tjumpsThresh0ld"}).Add("incorrect", { Username = "student" ; Password = "c#tjumpsThresh0l
ds"})
        let MOODLE_SITE = "http://46.101.125.192/moodle/"
        let MOODLE_USERNAME = "student"
        let MOODLE_PASSWORD = "c#tjumpsThresh0ld"
        let SEARCH_TEXT_2 = "Data Structures";
        let SEARCH_TEXT = "Business Process Management";
        let TEST_ENDPOINT = new Uri("http://127.0.0.1:4723/wd/hub") // If Appium is running l
ocally
        let INIT_TIMEOUT_SEC = TimeSpan.FromSeconds(300.0) (* Change this to a more reasonabl
e value *)
        let IMPLICIT_TIMEOUT_SEC = TimeSpan.FromSeconds(200.0) (* Change this to a more reaso
nable value - This is the time to wait when looking for an item on the page *)
        let cap = new DesiredCapabilities()
        static member val setupComplete = false with get, set
        static member val driver = null with get, set

        member this.Init ()=
            cap.SetCapability("deviceName", "LYO-L21")
            cap.SetCapability("appPackage", "com.moodle.moodlemobile")
            Actor.driver <- new AndroidDriver<AndroidElement>(TEST_ENDPOINT, cap, INIT_TIMEOU
T_SEC)
            Actor.driver.Manage().Timeouts().ImplicitlyWait(IMPLICIT_TIMEOUT_SEC) |> ignore
            Actor.setupComplete <- true
            log("Completed setup")
            this.connectSite()

        member this.connectSite () =
            let siteTxtBox = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1
]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.v
iew.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/android.view.View[
1]/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1]/androi
d.view.View[1]/android.widget.EditText[1]");
```

```
            siteTxtBox.Clear()
            siteTxtBox.SendKeys(MOODLE_SITE)
            let connectBtn = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1
]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.v
iew.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/android.view.View[
1]/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1]/androi
d.view.View[2]/android.widget.Button[1]");
            connectBtn.Click()
            wait 10
            //let (rtn:CompoundTerm) = null
            //rtn
            log("Site connected")


        member this.logIn (a:CompoundTerm)=
            log("Login started ...")
            let usernameTxtBox = Actor.driver.FindElementByXPath("//android.widget.LinearLayo
ut[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/andro
id.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/android.view.V
iew[1]/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1]/an
droid.view.View[1]/android.widget.EditText[1]");
            usernameTxtBox.Clear()
            let passwordTxtBox = Actor.driver.FindElementByXPath("//android.widget.LinearLayo
ut[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/andro
id.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/android.view.V
iew[1]/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1]/an
droid.view.View[2]/android.widget.EditText[1]");


            log("current user: " + string((a).[0]) + " with " + string((a).[1]))
            log((string)(a).Arguments)
            if (string((a).[1]) = "Password(\"Correct\")") then
                let studentLogin = STUDENT_CREDENTIAL.Item("correct")
                usernameTxtBox.SendKeys(studentLogin.Username)
                passwordTxtBox.SendKeys(studentLogin.Password)
            else
                let studentLogin = STUDENT_CREDENTIAL.Item("incorrect")
                usernameTxtBox.SendKeys(studentLogin.Username)
                passwordTxtBox.SendKeys(studentLogin.Password)


            let loginBtn = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1]/
android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.vie
w.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/android.view.View[1]
/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1]/android.
view.View[3]/android.widget.Button[1]");
            loginBtn.Click();


            let searchBtnXPathString = "//android.widget.LinearLayout[1]/android.widget.Frame
Layout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.view.View[1]/android.vi
ew.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[2]/android.view.View[1
]/android.view.View[2]/android.view.View[1]"
            let searchBtn = Actor.driver.FindElementByXPath(searchBtnXPathString)
            if searchBtn.Displayed  then
               log("Login Completed")
               Action.Create("login_Finish", (a).[0], LoginStatus.Success):> CompoundTerm
            else
               log("Cant complete Login")
               Action.Create("login_Finish", (a).[0], LoginStatus.Failure) :> CompoundTerm

        member this.courseSearch ()=
            let searchBtn = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1]
/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.vi
ew.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[2
]/android.view.View[1]/android.view.View[2]/android.view.View[1]");
            searchBtn.Click();
            let searchTxtBox = Actor.driver.FindElementByXPath("//android.widget.LinearLayout
[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android
.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.Vie
w[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/andr
oid.view.View[1]/android.widget.EditText[1]");
            searchTxtBox.Clear();
            wait (30)
            searchTxtBox.SendKeys(SEARCH_TEXT);
            let searchBtn = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1]
/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.vi
ew.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3
]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android
.view.View[1]/android.widget.Button[1]\n");
            searchBtn.Click();
            let (rtn:CompoundTerm) = null
            rtn

        member this.enrollForCourse ()=
            let searchResultItem = Actor.driver.FindElementByXPath("//android.widget.LinearLa
yout[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/and
roid.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view
.View[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]/
android.view.View[1]");
            searchResultItem.Click();
            wait(20);
            let enrolBtn = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1]/
android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.vie
w.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]
/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.
view.View[3]/android.widget.Button[1]");
            enrolBtn.Click();
            wait(20);
            let okConfirmationBtn = Actor.driver.FindElementByXPath(" //android.widget.Linear
Layout[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/a
ndroid.view.View[1]/android.view.View[3]/android.view.View[1]/android.widget.Button[2]\n");
            okConfirmationBtn.Click();
            let (rtn:CompoundTerm) = null
            rtn

        member this.addSubmission() =
            let courseButton = Actor.driver.FindElementByXPath("//android.widget.LinearLayout
[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android
.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.Vie
w[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/andr
oid.view.View[1]");
            courseButton.Click();
            wait(80);
            let topicButton = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[
1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.
view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View
[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/andro
id.widget.ListView[1]/android.view.View[3]/android.view.View[1]\n");
            topicButton.Click();
            let homeworkButton = Actor.driver.FindElementByXPath("//android.widget.LinearLayo
ut[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/andro
id.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.V
iew[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/an
droid.view.View[1]\n");
            homeworkButton.Click();
            let addSubmissionButton = Actor.driver.FindElementByXPath(" //android.widget.Line
arLayout[1]/android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]
/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.
view.View[3]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View
[2]/android.view.View[1]/android.view.View[2]\n");
            addSubmissionButton.Click();
            let textArea = Actor.driver.FindElementByXPath("//android.widget.LinearLayout[1]/
android.widget.FrameLayout[1]/android.webkit.WebView[1]/android.webkit.WebView[1]/android.vie
w.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[3]
/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.view.View[1]/android.
view.View[1]/android.view.View[2]/android.view.View[1]/android.view.View[1]/android.view.View
[1]\n");
            textArea.SendKeys("Hello World submission");
            let (rtn:CompoundTerm) = null
            rtn

        member this.checkIfExist (locator : string):Boolean =
           let elements = Actor.driver.FindElements(By.XPath(locator))
           elements.Count > 0
```