Automated test suites to test COVID contact-tracing apps

Vahid Garousi, Mark Lee
Queen's University Belfast, UK
{v.garousi, mlee24 }@qub.ac.uk

Abstract

More than 50 countries and regions have developed and published COVID contact-tracing apps, and most of those apps are open source. Some of those open-source contact-tracing apps come with small-scale automated test suites. But often times, due to time pressure, their development and test teams have not developed comprehensive automated test suites to test the functional behaviour of the apps. This project deploys a model-based testing (MBT) approach on two contact tracing COVID Android apps, using MBT tool named GraphWalker. By using MBT approach, the system is able to generate dynamic test-cases, which are executed on a real Android device using the Appium automation framework. The resulting end-to-end MBT test suites developed, demonstrated a "proof-of-concept" of MBT in Android domain, with notable benefits including an increase in overall test coverage and improvement in test-case design practices, resulting in less test code smells.

Table of Contents

Section 1 - Introduction and Problem Area

Section 1.1 – Automated Testing and its Challenges

Developing automated test suites for mobile applications can be a challenging and costly process, but failure to do so can result in error-prone apps [1]. Numerous studies [2, 3] have concluded that a lack of effective automated testing can have a detrimental impact on the user experience, leading to a high volume of complaints. Despite the clear benefits that automated testing presents, it is not always adopted by app developers when carrying out UI testing with many developers still following a manual testing approach. Recent data from a survey of 600 app developers [2] suggests, that time constraints and lack of exposure to testing tools was the biggest factors in the decision to use manual testing over an automated testing approach.

UI testing can be performed manually by a human tester, or it can run automatically, with the use of test execution tools such as Appium, which is commonly used in mobile applications. The setup cost of manual testing is usually minimal, but they are far less effective than an automated approach. There has been a large amount of research in the area, with most studies [1] reporting that manual testing is *"laborious, time-consuming, and error-prone"* and that Android applications call for *"scalable, robust, and trustworthy automated testing solutions."*

According to the MIT Technology Review [4], there are over 50 countries with Covid-19 contacting tracing apps, most of which have been developed under significant time-to-market pressures. This has resulted in the development and test teams for many Covid-19 apps being unable to develop comprehensive automated test suites to test the functional behaviour of their apps, as they are not granted the time or resources to learn automated testing tools.

In addition to the upfront costs of designing and developing good quality automated test suites, they require constant maintenance to adapt to changes made to the system under test (SUT), as it is updated. This is a significant investment but is vital to prevent software defects in applications due to ineffective testing and to ensure that production code is robust under many usage conditions. [5]

Section 1.2 – Approach to Automated UI Testing

The test-case design for the UI testing is usually a manual process, this can lead to a lack of test coverage and test-cases being ad-hoc, resulting in test smells. Test smell can be described as, "*sub-optimal design choices in the implementation of test code*" [5]. Many studies [5, 6] have observed that if your test suite contains test smells, then it will be less effective in finding bugs, with one study [5] finding 81% higher risk of being defective. This presents a false picture to developers and can result in premature release of code into production.

It is critical that automated test suites are designed and developed through a systematic approach to reasonably verify the SUT's functionality. As mobile applications have a large number of states and inputs in terms of UI, it is practically impossible to carry out systematic testing manually. Research on automated techniques for functional GUI testing of mobile applications [7], concluded that testing techniques which implement *"model-checking, symbolic execution, constraint solving, and search-based test generation approach tend to be more effective than those implementing random test generation."*

A systematic review of 83 primary studies on automated GUI testing of mobile apps [7] found that model-based testing (30%) was the most popular approach, followed by capture/replay (15.5%), model-learning testing (10%), systematic testing (7.5%), fuzz testing (7.5%), random testing (5%) and scripted based testing (2.5%). It also highlighted that 40% of these studies used automated testing techniques which designed GUI-based models for the SUT. The most effective automated test suites should be modular in nature [8], and the use of models allows for developers to adapt to changes in the GUI of the SUT, this is known as adaptive maintenance of test code [9]. The test suite should be systematically designed in order to achieve a high fault detection to code size ratio, ensuring the test suite is cost-effective.

The best automated UI testing approaches aim to increase test coverage, and to optimise model construction. The better optimised the models are, the more efficient testing tool they will be. This is a critical issue as most modern-day apps usually have complex GUI structures, which results in large models of high complexity [7]. In order for these large models to be readable, and maintainable, it is important to break down the entire system into several models, divided by distinct pages or app functionality.

Section 1.3 –Android applications with model-based testing

Android applications are made up of a series of screens called Activates. The Activates contain both the graphical elements the user can interact with and the back-end infrastructure that process the request after a user interacts with an element, for example, clicking a button. Android has a number of test automation frameworks which mimic the actions of users to test the functionality and potentially highlight areas that could be improved, some of which include Appium (appium.io), Espresso, UI Automator, Detox. The test automation test suites for the system developed for this project used the Appium framework which leverages Google's UiAutomator2 technology to facilitate automation on an Android device. Due to the nature of the Covid-19 contacting tracing apps requiring Bluetooth connectivity, all automation was carried out on a physical Android 10/11 device for this project. This was down to the fact that Android emulators do not include virtual hardware for Bluetooth.

For test-case design, the system used model-based testing (MBT) approach. MBT is a testing technique in which the SUT, "*is described with a formal model at such a level of detail that it can be used to automatically generate tests.*" [10] MBT commonly uses finite state machines to describe models and many tools provide graphing algorithms to traverse the models, therefore generating tests in a desired manner set by the developer and uses predefined model-level coverage criteria as the testing goal. The test-cases generated by the MBT tools are abstract versions and need to be combined with a test automation tool such as Appium, or Espresso (for Android applications) to produce executable tests which can be run against the SUT. MBT is categorised under black-box testing as the test suite is derived from the model of the front-end GUI components and does not interact with the source-code.

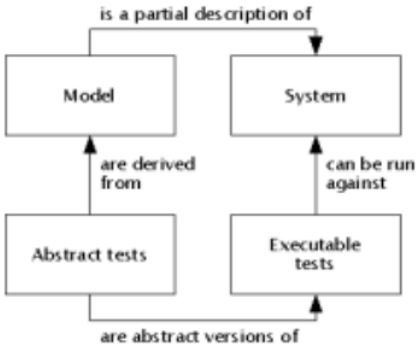


Figure 1- MBT simply approach

According to a recent study [11] MBT is suited to applications with, *"complex but decomposable behaviours*". The Covid-19 contact tracing apps fall under this category and using MBT will test realistic behaviours which could have been missed if the test-cases were generated manually. Furthermore, the abstraction level of model test-cases and execution allows for the reuse of execution actions with the same behaviour as they can be mapped to the same model action [12]. For example, a widely used action such as clicking a close tab, would only be declared once in the execution test-cases. However, when testing using MBT, it is used in multiple different states in many different sub models.

Most Android applications are developed without using MBT, with one research paper indicating that the adoption of MBT has been affected by organisational difficulties and the lack of easy-to-use tools [10].

Section 1.4 – Needs and motivations for the project

In the context of Covid-19 contact tracing applications this paper will report on the experiences of choosing and applying MBT approach on the following Android applications:

- NHS Covid-19 (England and Wales)
- Protect Scotland (Scotland)

The features of the two SUTs that will be tested, can been seen below in Table 1

Table 1 feature comparison between two SUT

	Core feature: recording contacts and notifying them when this user enters positive	**Add test result**	**Book a test**	**Symptom checker**	**Share and protect: option to share app with others**	**Venue check-in - QR code scanner**	**Isolation countdown**	**Show number of times app downloaded**
UK: NHS COVID-19	**x**	**x**	**x**	**x**		**x**	**x**	
Protect Scotland	**x**	**x**	**x**	**x**	**x**			**x**

For this project, the models designed would be unique artifacts as they would be designed from scratch. The main goal of GUI testing of Android apps is to thoroughly test its functionality and validate its behaviour by enforcing various user interactions. These user / system interactions are represented on the behavioural models. Examples of behaviour models used for test-case generation include, Finite State Machine (FSM), Unified Modelling Language State Machine (UML), State Charts, Markov Chain Models and UML activity diagrams [13].

The most widely used MBT approach in Android applications is Extended Finite State Machines, due to the apps being stateful in nature. EFSM are based of FSM with significant enhancements, to include *actions* which can occur on nodes or edges and *guard conditions* which controls the flow to the edge they are applied toon System sample [14]. Figure 2 shows an example of UML State Machine which includes a total of 5 nodes (states) and 12 edges (transitions).

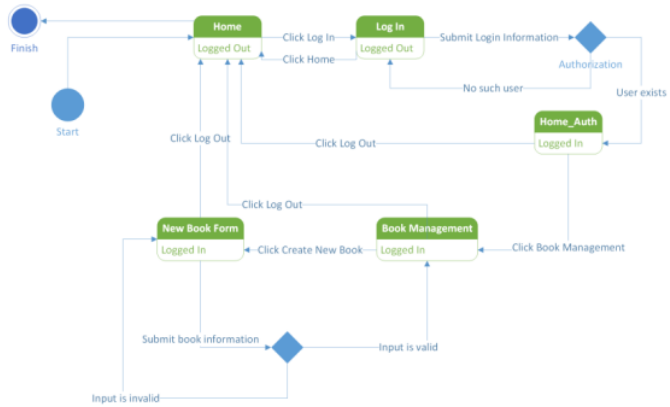


Figure 2 - UML State Machine for Library Information System sample [14]

After reviewing the research studies of GUI testing methods [10, 15], it was decided that MBT was the best approach in order to develop a comprehensive automated test suite in terms of GUI testing.

Section 2 - System Requirements and Specification

For the project to be successful, the system developed should be able to carry out an end-to-end functionality testing of GUI using MBT for each of the two covid contact tracing Android applications, outlined in the introduction. The aim is to simulate real-world conditions as much

as possible before starting any tests. This includes carrying out the tests with a variation in network conditions. The system will use an MBT approach to test the core functionality of each app and be able to report back which UI element or user action caused the failure during the test. This was developed in three main phases:

1. Designing and developing GUI flow models / EFSM for each SUT. This will be done firstly by GUI flow tools, later implemented using MBT tools which can be executed offline (not connected to SUT and without automation code). The models will be made of nodes (pages / app views) and edges (actions – both user and system)
2. Extending on the first stage, will be a suite of automation methods / classes which will execute the actions directly with the SUT's. This will provide the system the ability to perform online MBT as it connects directly to the Android device via Android driver, mimicking user actions as described by the abstract test cases produced in part 1.
3. The final stage will be the reporting, showing a statistical breakdown of tests, such as edge coverage and number of times visited for each node.

The scope of the system is constrained by a number of environmental factors:

- All test development will use a single Samsung Galaxy S10 Android device, with testing occurring for both Android 10 and 11 versions.
- This device will be physical (unable to use Android emulator due their lack of Bluetooth virtual hardware).
- Both NHS Covid-19 (App version 4.2) and Protect Scotland (App version 1.2.1.81), updates will be turned off, as to prevent updates *"breaking tests"*. In an ideal world you would work alongside a development team when designing and developing MBT test suites, however this was not feasible for this project.
- Test runs will be executed under WI-FI conditions, which is prone to variations and occasional down time.

Section 2.1 - Use Cases

The three high level overarching components of the system described above can be broken down into individual use cases, which are the building blocks of the overall system. Figure 3 showcases a visual example of requirements via a use-case diagram. It is clear from the use-case diagram that the level of abstraction varies. The initial generated test cases derived from designing and validating the GUI flow models focus only on nodes and edge inputs / outputs,

with some conditional logic to represent the current state of the application as described in section 2. For the system to operate in a streamlined, cohesive manner it is important to use a model-based testing tool which can eloquently interlink the system requirements. Section 2.2 will look at the MBT tool selection process.

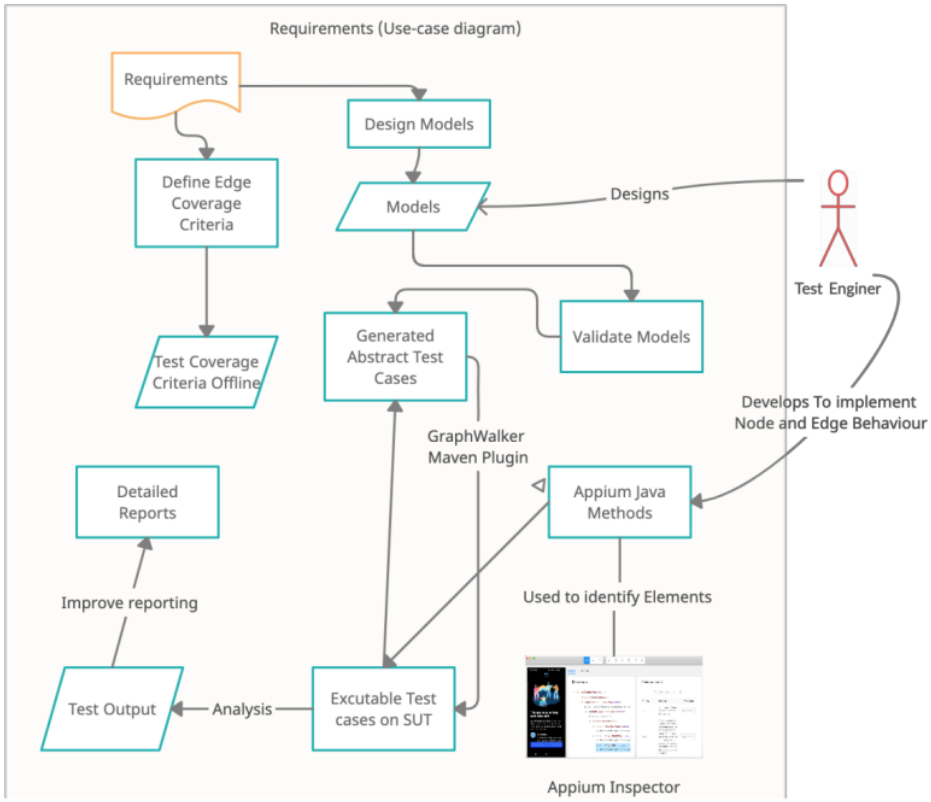


Figure 3- Requirements (Tests-case diagram)

Section 2.2 Mode-based approach and tool selection

When it came to deciding which automation framework and MBT tool to use, a number of factors where considered, with the most important criteria being: (1) the tools cost – ideally the tool used would be open-source (therefore free to use), (2) the tool would match the test requirement of GUI testing of Android apps, (3) ease of use - with appropriate documentation and sufficient features to be able to design the abstract test cases and implement the executable test cases using an automation framework such as Appium.

Due to time constraints, it wasn't possible to test all the tools, but the selection was narrowed down to opensource tools which left a standout option, GraphWalker (https://github.com/GraphWalker/graphwalker-project/wiki). It matched the outlined criteria, as it was open sourced, with well-structured, detailed documentation. It also provides a user-friendly modelling tool for designing abstract models using EFSM, which has two execution modes, online and offline. Offline mode means generating a test sequence once that can later be run automatically. This can be valuable to test that the model, along with the path generator(s) and stop condition(s), works. Whereas online mode connects directly with the SUT and tests it dynamically. GraphWalker will start either as a WebSocket (default) or a HTTP REST server, to visualise the test progression. Furthermore, GraphWalker tool includes various maven plugins, enabling seamless integration to the systems Java-based project, which has led to the system implementing the GraphWalker's maven architecture to help fast-track development.

In researching relative work within the area, serval industry case studies that used GraphWalker [8, 16] were reviewed. This displayed the tools viability in a real-world industry practice. One of these projects was developed by Dr Vahid Garousi, (https://github.com/vgarousi/MBTofTestinium), which applied GraphWalker within the web application domain and provided an experience report on its findings [17]. These case studies have helped with the learning curve that MBT and the GraphWalker tool entails.

Section 2.3 – Requirements of the automated test suite

Functional requirements

- Test suites use MBT to run end-to-end test automation using GraphWalker with a configurated generator and stop condition for each Covid-19 contact tracing app
- The models designed cover 100% of the GUI elements for each app and is executed with the requirement of 100% edge coverage, meaning the path generator will generate a new step in the path until the stop condition is fulfilled
- The test models should be of a high level of abstraction as to allow for potential of both Android and iOS implementations
- During execution the user can visualise a live feed of the model execution using GraphWalker studio connected to the system via the WebSocket protocol.

- Detailed excel log reports will be produced automatically after each MBT test execution. These will provide a full breakdown of each test run, including test path sequence, entity (node or edge) and model visit count and duration.
- An email will be sent automatically after each MBT test run of each SUT. This will attach the excel log report and the basic report produced by GraphWalker.

Non-Functional requirements

- Useability: the system is easy to set up, with minimum set up time required, with simple easy to follow installation instructions.
- Scalability: the system follows design practices which can handle Android apps which are of high complexity, and require large scale MBT flow models.
- Configurability: the system should be highly configurable with the ability to change the test path generation algorithms and edge coverage requirements through a config file. For example, changing the test generation used by GraphWalker from random to weighted random.
- Reusability / Modularity of test code through following the "software test-code engineering" (STCE) process [18]: The Java Appium methods developed can reused for both applications and used for large number of test cases. For instances the checkElementVisible method is used in over 70 different test cases when verifying the state of the application at nodes.

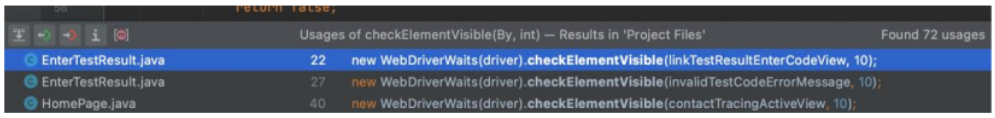


Figure 4 Example of reusability within the system

Section 3 – Design

This section will include an overview of the phases of work and the decisions made when designing an MBT system for the SUT's outlined. It was vital that the system was designed using best practices and based on the selection of GraphWalker tool in Section 2.2 Mode-based approach and tool selection, was complementary to Android mobile applications.

Section 3.1 – Test Model Design

To begin with, an initial model for the Protect Scotland app was designed. It used a simple tool to design the model (FlowMapp). The first step when designing the initial model included covering each page (node) in the application and mapping every possible transition to other pages / activities within the application for each node. These transitions between nodes are known as edges. In order to visualise the flow of the UI, the initial model used screenshots of each page / activity to represent the nodes, and arrows to represent the edges.

Figure 5 Snippet of Protect Scotland initial model designed

For large-scale apps with high UI flow complexity, it can become overwhelming and difficult to understand as the number of transitions between nodes and edges increase. As you can see in Figure 6 it can be difficult to follow and completely understand the model. Therefore, it would benefit from being broken down into separate models, which are distinguished by app activity or functionality of a particular section within the app. For example, the onboarding stage could be its own model.

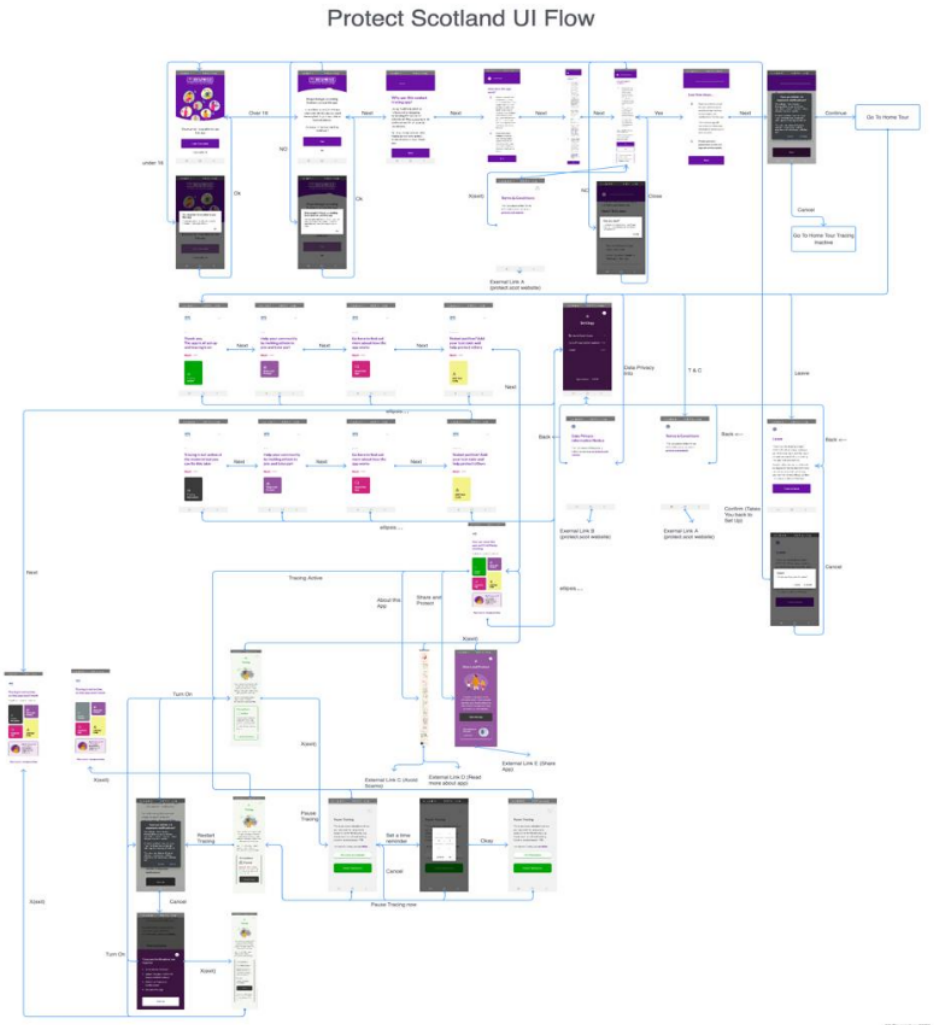


Figure 6 Protect Scotland UI flow Diagram

Research has indicated that it tends to be preferable to design test models via automated means [19, 20]. This results in test models which have a low level of abstraction and provides the benefits of efficiency without the need for human interaction. However, as the SUTs are only two Android apps; it allows for higher level of abstraction if they are designed manually without the drawbacks of the lack of efficiency. Manually designed models will provide a more realistic reflection of the user interaction with the SUT, whereas automated creation of models would be more granular with more technical UI language used to describe the test model's behaviour.

Using a higher-level of abstraction also provides more digestible and understandable design / documentation of each application's front-end experience which is akin to black-box testing, as it models the features of the application and does not account for how these features are implemented "*under the hood"* (back-end implementation). Furthermore, using a high level of abstraction would provide the future possibility of implementing MBT of each app on the iOS platform.

When it came to designing models using GraphWalker Studio, the design principals used was based on findings from various research papers on GraphWalker [8, 21]. In a recent 2020 paper GraphWalker founder Stefan Karlsson reported that models which represent an Android app should be distinguished by feature, meaning each feature within an application should have its own model to represent its behaviour. This backed up the observations when designing the initial test model for the Protect Scotland app. Therefore, its *good MBT modelling practices* for an overall model of an application designed using GraphWalker Studio tool to be represented by various sub-models which are in linked via *shared nodes*. Figure 7 is an example of a sub

model made using GraphWalker which represents the *Review Symptoms* functionality within the NHS Covid-19 app.

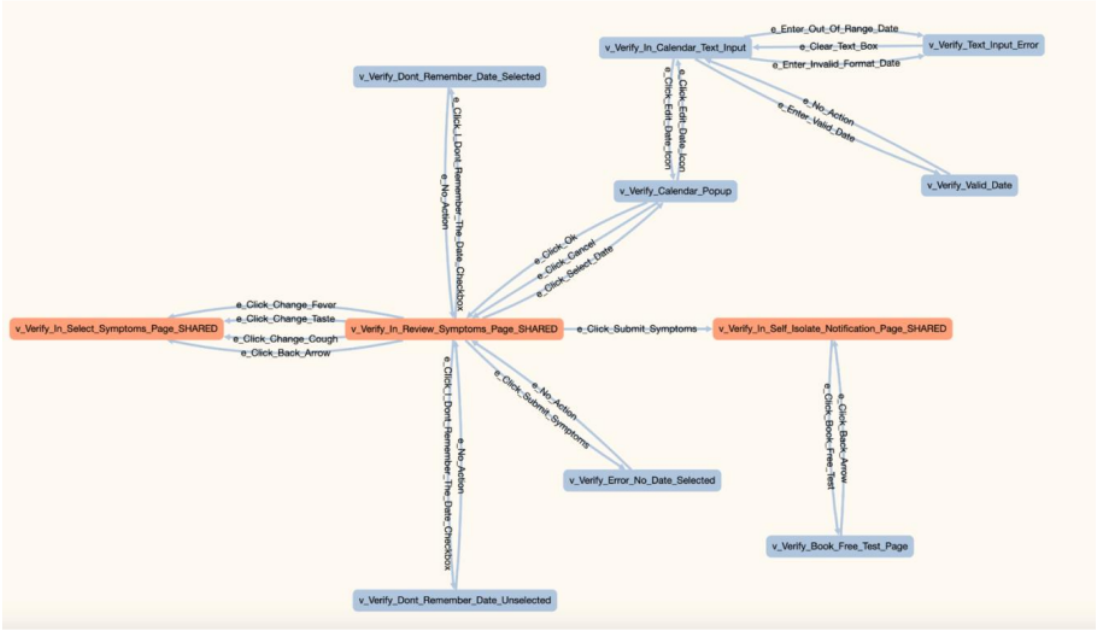


Figure 7 Example of sub-model in GraphWalker studio

Model components / configuration

When designing the MBT test models using the GraphWalker Studio tool, there were a number of features to consider, which provided an added benefit to the overall model design. A large number of parameters / configurations are provided by GraphWalker, which were modified during the design process of the MBT test suites. The following is a brief overview of some GraphWalker model design configurations (more information can be found on GraphWalker's official documentation):

- Nodes: Represent the current state of the application, these states are verified through asserting if one or more expected UI element is present using functions created with the Appium automation framework and asserting if they are present (using JUnit), e.g., `v_In_Home_Page_SHARED`
- Edges: The transitions between nodes which occur during a user interaction with the app. The most common edges are button clicks. All edges mimic the actions of a user via Appium automation framework, e.g., `e_Click_Back_Arrow`

- Start Element: The node within the model which the MBT test suite starts from. In the case of Android apps this is the `appActivity` of the `appPackage`.
- Actions: These are optional and can be applied to a node or an edge within the model. Actions usually change a variable value, as you can see in Figure 8 the `cameraPermssion` flag is set to *true* whenever the `e_Click_Allow` edge is traversed during the MBT run.
- Guards: Each SUT models can include guards (Booleans) to allow the test cases to be executed with accordance to the current scenario / state the app is in. Guards are set by actions.
- Weights: These are applied to edges only and they represent the probability of an edge getting chosen. In order for the *weight* to be used the path Generator should be set to *weighted_random*, or else the *weight* set will be ignored.
- Stop conditions: These are an essential aspect of MBT design, they decide when MBT test execution comes to a halt. GraphWalker studio supports serval stop conditions, including, *edge coverage*, *vertex coverage*, *time duration*. Generator and Stop conditions implementation will look in more depth on the decision-making process when deciding upon the most optimal *generator* and *stop condition* for the two SUTs in terms of overall performance (i.e., fault detection effectiveness), but also in terms of a realistic execution time which is not too long.

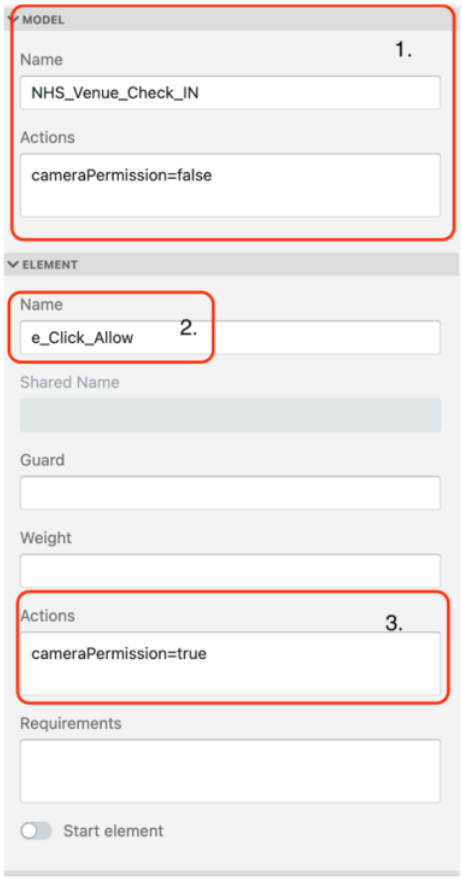


Figure 8 Show Properties applied to Click Allow edge in NHS_Venue_check_In model

Section 3.2 - Appium Automation design

After a design plan was in place for designing the test models, the next step was the actual automation of the Android SUTs. Based off the requirements highlighted in Section 2, it was decided that Appium automation framework would be the best solution.

UI Element Selector

Through using the automation framework Appium, each UI element can be located through various selectors similar to *div id* in front-end web browser applications. As the two SUTs are Android application, the system used the Appium Desktop Inspector to provide a visual representation of the SUTs at certain stages via screenshots and the source code of the front end which contains the UI elements.

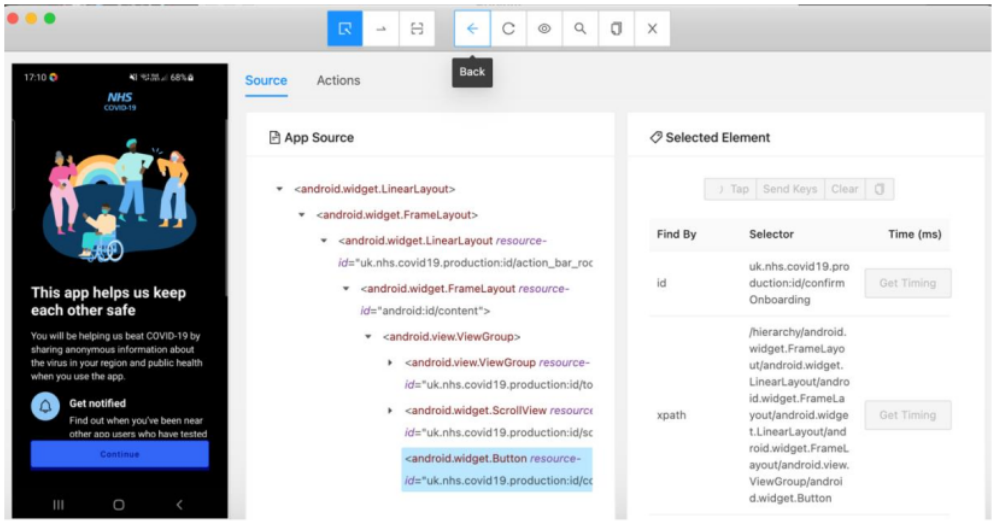


Figure 9 Example of Apium Desktop Inspector

Appium with *Android UIautomator2* driver provides various locators used to find a UI element. According to Appium documentation `ID` (resource-id) is the preferred method to find elements, followed by `Accessibility ID` if `ID` is not available for said element. Other options include `xpath` (not recommended, has performance issues) and `Class name` (full name of the UIAutomator2 class).

All Java Appium methods are designed, with a hierarchy approach when it comes to deciding upon which locator to use:

1. `resource-id / ID`
2. `Accessibility ID`
3. `Class name`
4. `Xpath`

```
private By exposureNotificationReminderContainer =
        MobileBy.id("uk.nhs.covid19.production:id/exposure_notification_reminder_container");
```

Figure 10 example UI element finder by ID

Android base Test Class

An Android base test class was designed, to allow for each SUT models to extend upon. This allowed for uniform implementation of test models and set up the configuration properties of the test. These configuration properties included:

- App package (unique application ID which identifies the app on the Google Play store)

- App Launch Activity (the page which is launched when the app is opened)
- Device ID (this is backup configuration, as ID is normally scraped from connected USB device via an ADB command)
- Appium Server Port

This class sets up the Android Driver, with the configured properties, which in turn starts an Appium Server and connects to it. This helps achieve the non-functional requirement of *configurability* which was part of the requirements highlighted in section 2.

Section 3.3 – Overall Architectural Description of System

In order to provide a better understanding of the overall system design, an architectural overview has been designed to allow for visual representation of the system at various levels of abstraction.

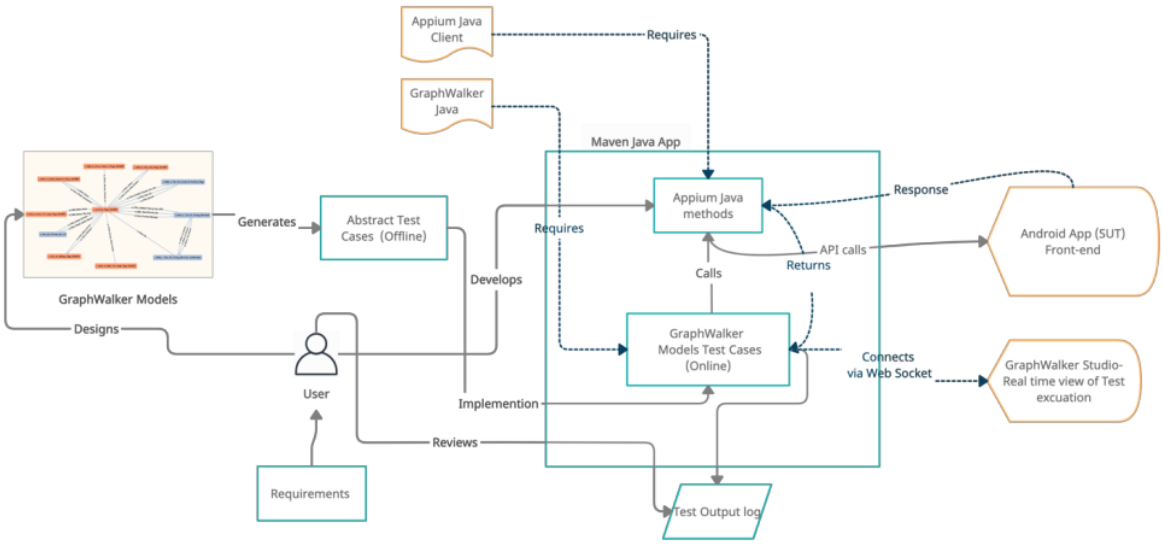


Figure 11 Architectural structure of Java System Design

As you can see from Figure 11, the test engineer uses the system requirements to design the GraphWalker test models, which generates abstract test cases which can be executed in *offline mode* (generating sequences without the SUT running), enabling the test engineer to test whether the test models designed are valid and that the generator and stop condition work as expected. The test engineer then develops the Appium Java methods and configurates all the

infrastructural requirements for the test suite execution, a graphical breakdown of this can be seen below in Figure 12. The Appium Java method requires an *Appium java-client* package, to implement edge and node behaviour when test suites are connected directly to the SUTs. The actual automation will be performed by an Appium server which will make API calls to the SUTs (which is connected to test device via USB), which will respond with the outcome if the edge or node behaviour was performed successfully or not. Lastly, as shown in Figure 11, the online execution uses a test runner and connects to the GraphWalker Studio web application (via WebSocket) to allow for live visualising of MBT progress using the test-models as means to visualise the traversal of the test models.

To understand how the system will work, it is important to have a low-level view of the system, to see how the different components work together. Below is an overview of the software system, including an outline of the Java maven structure:

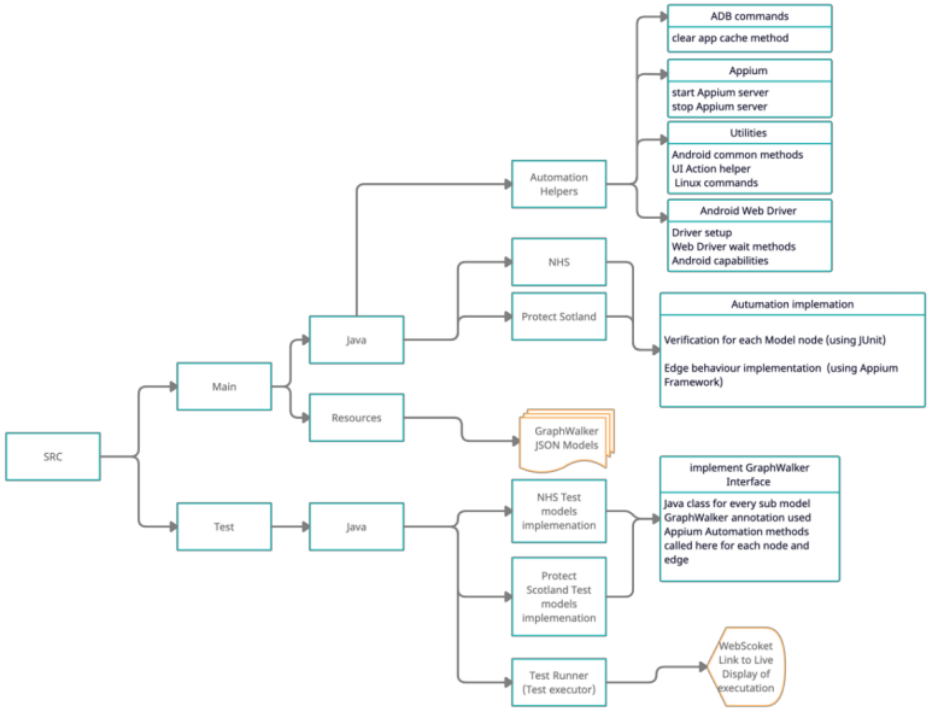


Figure 12 Overview of Software System Design

The software design follows a typical maven project layout with some differences in order to combine GraphWalker maven architecture principles. All Java dependencies will be listed in the *pom.xml* which will be downloaded dynamically from the Maven 2 Central Repository. The GraphWalker JSON models will be stored in the resource folder, which will be used to generate the abstract test cases when running the following command:

```
mvn graphwalker:generate-sources
```

These abstract interfaces generated are implemented in the *src/test/java /<app>model* folder. It is these extended test suites that GraphWalker (using Appium) will run against the SUT's. Each node and edge automation of the front-end behaviour will be implemented in *src/main/java /<app>* and called when required in *test* folder.

The test runners are located in *src/test/java/tests,* which is where the GraphWalker test is kicked off. Each test will start a WebSocket server on port 8887 to provide a live visualisation of the test execution using GraphWalker Studio.

UML Class Diagrams

Unified Modelling Language class diagrams are used in the design process to describe the structure of the system by showing the systems' classes, their attributes, operations (or methods), and the relationships among objects.

The following are examples of UML Class diagrams created for the system using the PlantUML tool (plantuml.com):

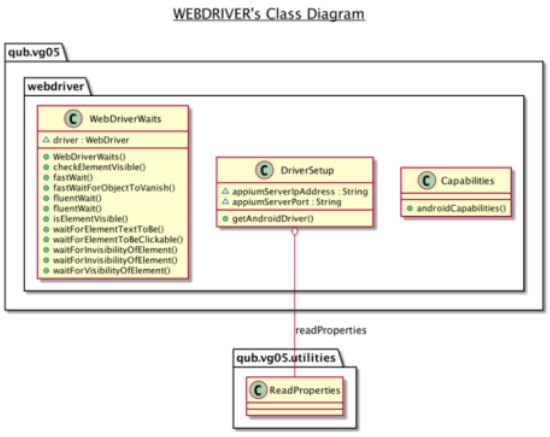


Figure 13 Webdriver class diagram

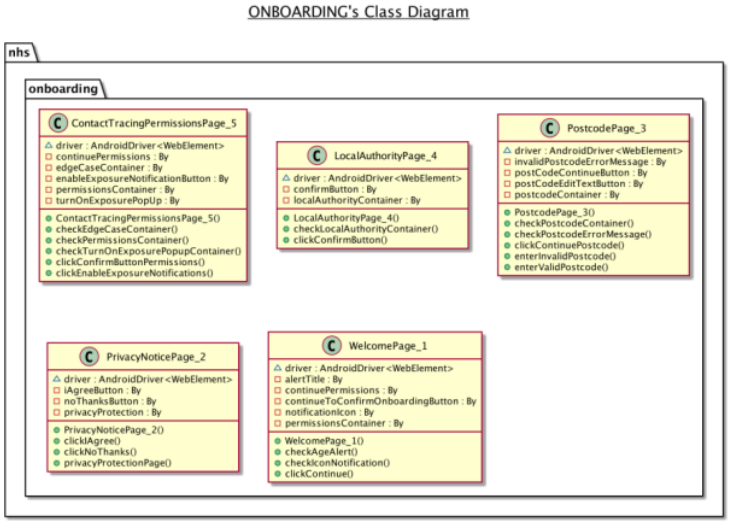


Figure 14 Onboarding Class Diagram

Section 4 – Implementation

With the design phase complete, the next stage in the software development lifecycle is to implement the system based off the design decisions that have been taken in

Section 3 – Design. In line with the structure used in designing the system, the test models will be implemented first. This section aims to breakdown how each component of the system was implemented in terms of languages, libraries and tools used.

Java 8.0 was chosen as the programming language and the project used Maven to manage Java libraires and automatically build via Maven's *pom.xml*. Java was chosen due to its integration capabilities with GraphWalker and Maven provided a simple build and dependency management solution. GraphWalker provides various Maven dependencies and plugins which were beneficial in reducing the setup time. These include: `graphwalker-io`, `graphwalker-core`, `graphwalker-java`, `graphwalker-maven-plugin` and `graphwalker-websocket`.

The system was developed using JetBrains IntelliJ IDEA (Ultimate Edition), as it provided out of the box support and features for all of the project's needs. For instance, IntelliJ supports a

fully functional integration with Maven and other features such as the ability to run the MBT test suites without the need for third party plugins or extra configuration.

Section 4.1 - Test Models Implementation

Due to the nature of the system, it was essential to start with the implementation of the test models for each SUT. As discussed in

Section 3.1 – Test Model Design, the test models were designed using the GraphWalker studio tool, via sub-models, which are grouped based on a distinct feature / functionality within the Android applications.

The MBT test models were implemented to address; user flow, node verification, changes in application state, error checking and UI inputs in each test model [22]. The following are the key implementations made to the MBT test models for both SUTs:

1. User Flow: transition from one node (page or state) to another through an edge (this can be an action such as click x button).
2. Error checking: Node to itself (invalid input, for example entering an invalid test code)
3. State verification: Checking the current application state, matches the expected state of the application.
4. Changes in application state: recognising application current state and able to react accordingly. For instance, if one of the SUTs has contact tracing turned on, then the edges available to it will be restricted to reflect its current state. *Actions* and *guards* are the predominant methods used to apply state to the overall MBT test suites in order to reflect the current state of the SUT, as discussed in Model components / configuration section. The state of the applications and consequently the test suites are fluid, meaning it can change depending on the path the generator takes.
5. Ui inputs: For the two SUT the UI input which are required comes in form of text inputs, for example `e_Valid_Postcode`, is an edge which enters a valid postcode into a textbox.
6. End-to-end test automation using GraphWalker parameters to configurate the generator and the stop condition for each SUT.

The following is an example of the test models implementation of the first SUT (*NHS Covid-19)*. In total there were 12 models designed and developed (2 of which can be seen in Figure 16), with 29 *shared* Nodes linking the models together. Altogether those 12 models had 71 nodes and 138 edges (total of 209 entities). Figure 15 shows two screenshots from the SUT: the first is the launch screen (start of onboarding process) and the second is the *home* activity screen which is shown after onboarding is complete.

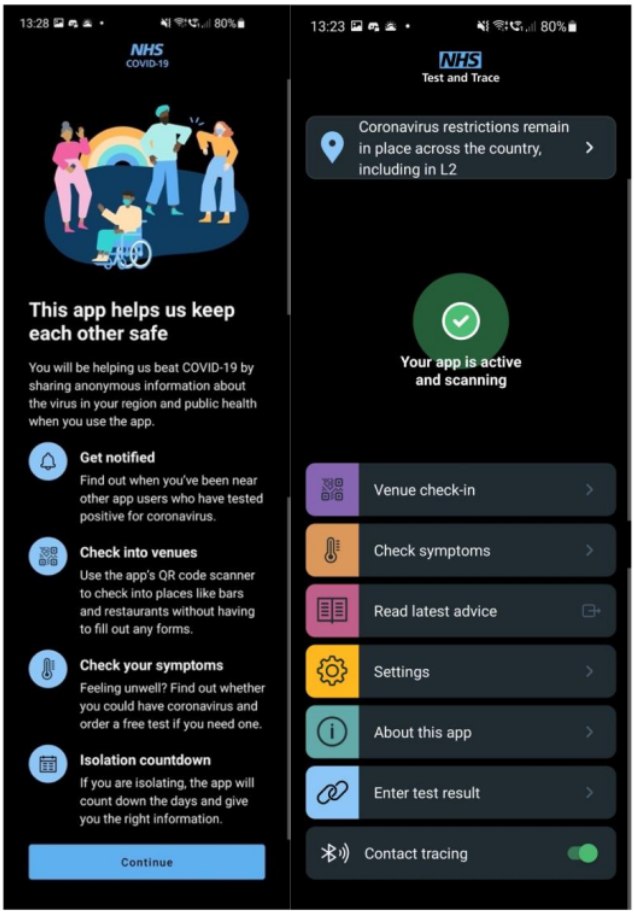


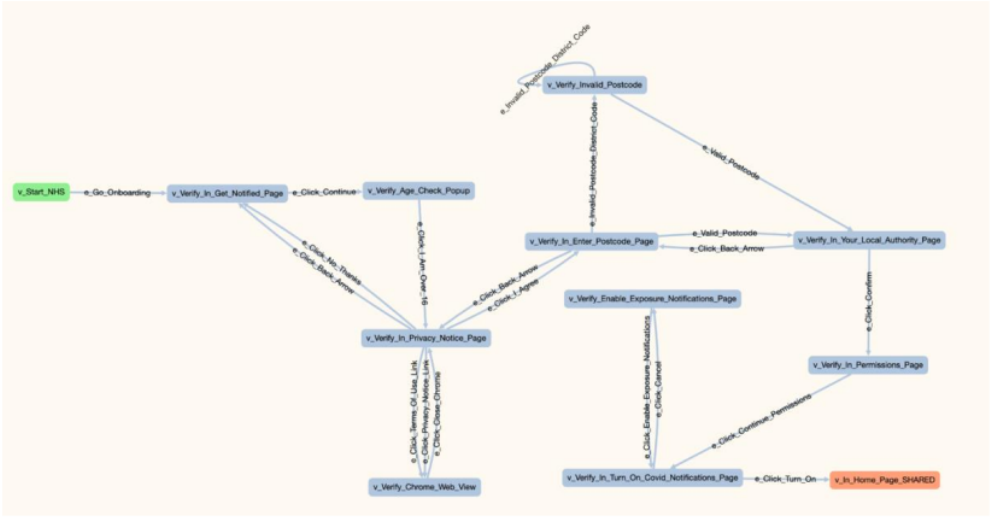
Figure 15 screenshots from the SUT - NHS Covid-19

Figure 16 shows two test models designed for testing the above activity screens within the NHS Covid-19 app. As shown from the models, each edge corresponds to an action, e.g., `e_Click_Enter_Test_Code`, (in the *NHS_home* model) and each node corresponds to a current state of the application, which is verified via assertions of the presence of one or more UI elements, e.g., `v_Verify_In_Venue_Check_In_Page_SHARED`. The MBT behaviour model that

was used to represent the SUTs using GraphWalker was EFSM, which is the most popular approach when modelling Android applications, and essentially represents all the possible UI flows available to a user of the app.

The system is able to jump between nodes on different sub-models via *shared* nodes. Nodes which contain the *shared* annotation (which are highlighted in orange in Figure 16), have corresponding nodes in one or more sub-models which share the same tag, so when MBT test suite is running and visits a *share* node it can jump to another node which has the same *shared* tag. This provides an eloquent way to break the entire SUT into several feature specific models.

The start node `v_start_NHS`, is highlighted in green in Figure 16 and as the name suggests it is the element which the MBT execution starts on. To set a start element in GraphWalker Studio, you slide the toggle on the element of choice.



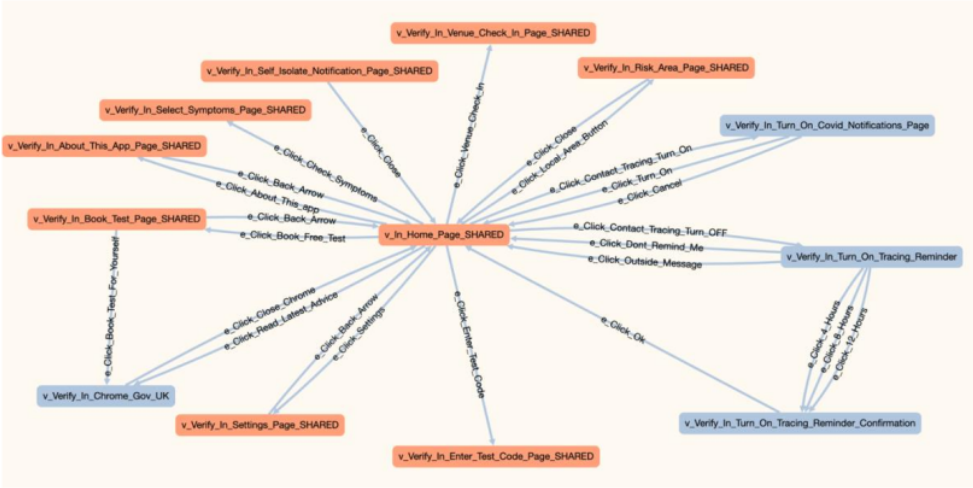


Figure 16 Two MBT test models for the SUT (NHS Covid-19): the MBT models for onboarding and home app activities (shown in Figure 15)

The Protect Scotland model design and implementation resulted in total of 11 models, with 69 nodes, and 114 edges (total of 183 entities). In order to interlink the 11 test models, 29 *shared* nodes were used across all the sub-models. Figure 17 shows 2 out of the 11 sub-models that make up the Protect Scotland. As you can see in the *home tour* test model (see below), there are two possible states which the app user can be in according to the path the random generator traverses. The two states occur when *contacting tracing* is turned *on* or *off*. When contacting tracing is turned on the test execution goes through v_Verify_In_Home_Tour_Active_Page_SHARED shared node from *home tour* model to the *home page* model.

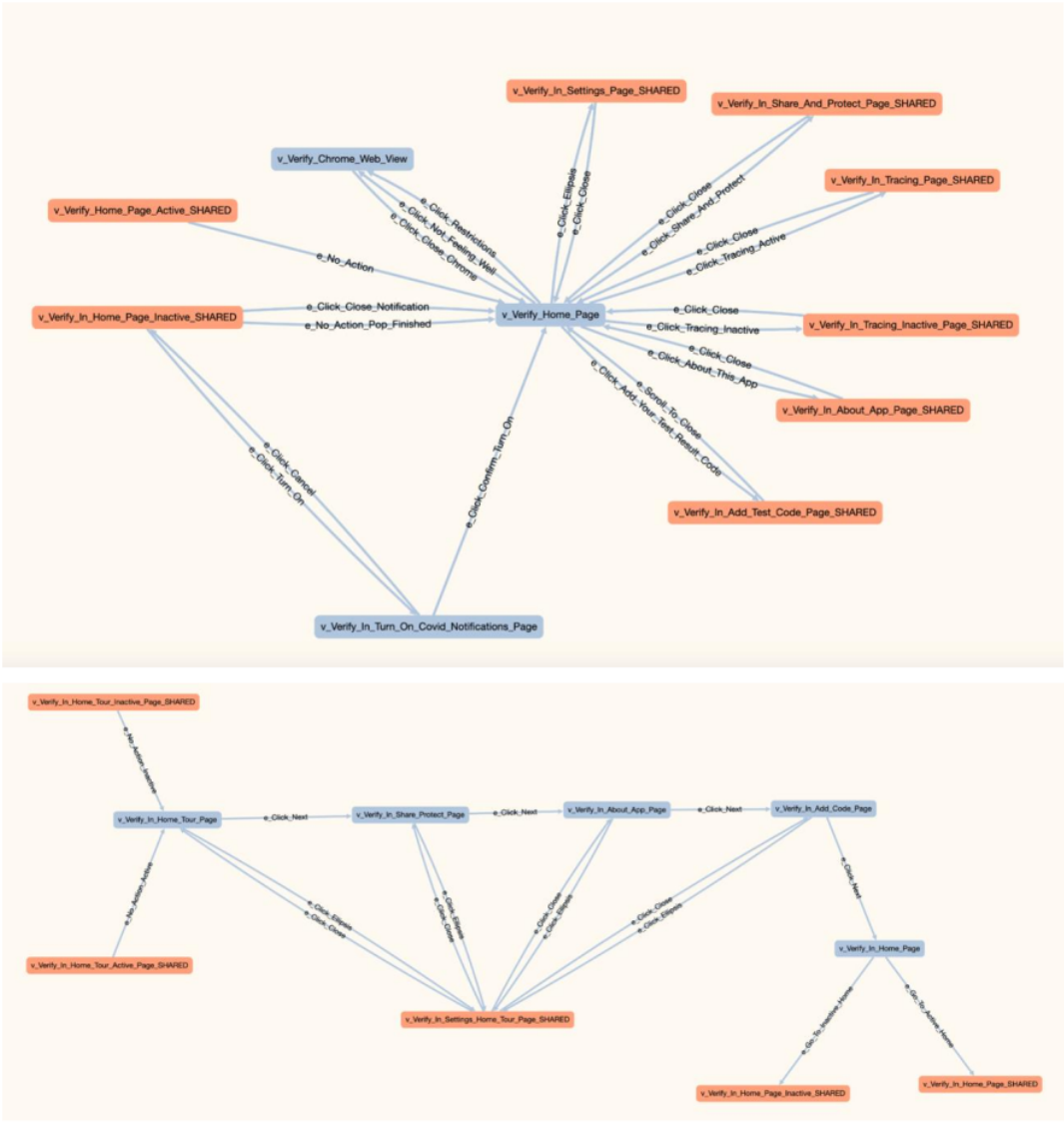


Figure 17 Two MBT test models for the SUT (Protect Scotland): the MBT models for home tour (bottom) and home page (top)

Action and Guard implementation

Both SUTs applied *actions* on both nodes and edges within the test suites in order to reflect the current scenario of the application. For example, in the NHS Covid-19 test suite, there is an action applied within the `NHS_Home` model, when the `e_Click_Turn_On` (Figure 18 - 4) edge has been activated during execution, the `tracingActive` is set to true (Figure 18 -5) to reflect the change in state of the application. As shown in Figure 18 -1, `tracingActive` is set to true by default. The variable `tracingActive` is applied as a *guard* on the

`e_Click_Contact_Tracing_Turn_On` (Figure 18 - 2) edge where the edge can only be taken if contact tracing is turned off which is represented by `!tracingActive` (Figure 18 -3), hence the guard blocks the edge from being taken when contact tracing is on (you cannot turn contact tracing on if its already turned on).

Figure 18 Examples of Actions and Guards implementation

Validating test models

To ensure the test models were properly designed, they were tested offline, with a range of model generators and stop conditions using GraphWalker studio. This process was fairly straight forward, as the test suites were not actually connected to their SUT, therefore the test execution delay could be adjusted to either speed up or slow down the test execution.

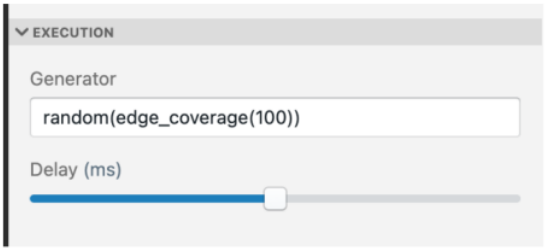


Figure 19 GraphWalker offline execution configuration

The final step to ensure the test models where "*production ready*" was via the GraphWalker Maven command:

```
mvn graphwalker:validate-models
```

Section 4.2 – GraphWalker Java implementation

After developing the models for the SUTs the next step was to integrate them into the Java based project using the GraphWalker maven plugins and libraries mentation at the start of Section 4 – Implementation. In order to read the JSON files which are outputted by GraphWalker studio of the test models, the GraphWalker java maven dependency `graphwalker-io` was used via the maven command: `mvn graphwalker:generate-sources` to generates interfaces from models that are placed in the folder `src/main/resources`. The interfaces are generated in the target directory (*target directory is used to house all outputs of builds in maven*) as you can see below:

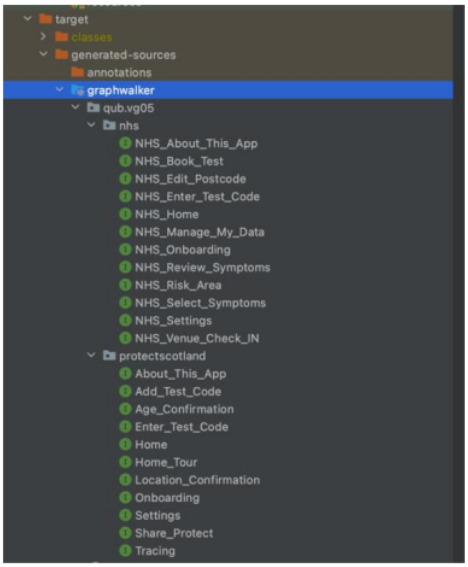


Figure 20 generated interfaces which programmatically represent test models

The interfaces created are abstract classes, which contain the abstract test models, which then are implemented using Appium Java methods to perform the actions on each node and edge within the MBT test suites (see Section 4.3 – Development of nodes/edges' behaviour using Appium). The interfaces use Java annotations provided by GraphWalker: `@Model, @Vertex and @Edge`, to represent the models *JSON file, edges and nodes* respectively. Figure 21 is an example of part of the *NHS_HOME* abstract test cases, which can be seen below.

```java
// Generated by GraphWalker (http://www.graphwalker.org)
package qub.vg05;

import ...

@Model(file = "qub/vg05/NHSModel.json")
public interface NHS_Home {

    @Vertex()
    void v_In_Home_Page_SHARED();

    @Vertex()
    void v_Verify_In_Enter_Test_Code_Page_SHARED();

    @Vertex()
    void v_Verify_In_About_This_App_Page_SHARED();

    @Edge()
    void e_Click_Dont_Remind_Me();
```

Figure 21 Abstract classes including GraphWalker Java annotation

The next stage is to implement the interfaces, with each class extending the `AndroidBaseTestClass` (role was described in more detail in Android base Test Class), which applies the GraphWalker annotation `@GraphWalker`, used to label a class as a GraphWalker test. It was vital to be able to separate test suites by app and this was achieved by the *groups* parameters within `@GraphWalker`, which allowed for the MBT tests to be run with a parameter option set to reflect which app's test suite you want to run. The parameter is applied to the GraphWalker test command is *-Dgroups=<app>:*

- NHS: `mvn graphwalker:test -Dgroups=nhs`
- Protect Scotland: `mvn graphwalker:test -Dgroups=protectScotland`

The path generator and stop condition are set by the *value* parameter and if the sub-model implementation contains a start element, it is set with the *start* parameter. Let's take a look at Figure 22, which showcases an example of GraphWalker annotations for both SUTs (Onboarding -NHS and Setting -Protect Scotland):

```
@GraphWalker(value = "random(edge_coverage(100))", groups= "protectScotland")
public class SettingsModel extends AndroidBaseTestClass implements qub.vg05.Settings {
```

```
@GraphWalker(value = CoverageValue.RandomEdgeCoverage100,start = "v_Start_NHS", groups= "nhs")
public class OnboardingModel extends AndroidBaseTestClass implements qub.vg05.NHS_Onboarding {
```

Figure 22 Is an example GraphWalker annotation need to run a class as a GraphWalker Test

Section 4.3 – Development of nodes/edges' behaviour using Appium

Appium and Android Driver setup

As stated previously, the online execution (i.e., connecting directly to SUT) of the MBT test suite was developed using the Appium automation framework in order to implement the abstract test cases. Through using Appium's Android driver, the system connects to the SUT via USB connection (Android device requires *developer mode enabled*, and *USB debugging enabled*), which requires an Appium server to be started, ADB command line tools installed and Android capabilities to be configured. This is achieved through using the @BeforeExecuation GraphWalker annotation which represents the set-up phase before GraphWalker test suite execution starts. All configurations ae stored in the `config.properties` (see Section 7 – Appendix), allowing for a highly configurable system. Figure 23 shows the NHS-Covid-19 setup function, which starts an Appium server, initiates the Android driver, before activating the driver to launch the application.

```
@BeforeExecution
public void setup() throws IOException, InterruptedException {

    appiumServer.startServer();
    appPackage = readProperties.getProp("nhsAppPackage");
    appLaunchActivity = readProperties.getProp("nhsAppActivity");

    initiateDriver( systemPort: "systemPort", appPackage, appLaunchActivity, noReset: true);
    new AdbCommands(driver).clearAppCache(appPackage, clearAppCache: true);
    TestLogOutputConfiguration.appVersion = new AdbCommands(driver).getAppVersion(appPackage);
    TestLogOutputConfiguration.androidVersion = new AdbCommands(driver).getDeviceVersion();
    TestLogOutputConfiguration.deviceName = new AdbCommands(driver).getDeviceManufacturer() +

    driver.activateApp(appPackage);

}
```

Figure 23 Set up Appium Android Driver and Appium Server before MBT execution

This setup logic is only required in the model implementation which contains the start element (for NHS Covid-19 this is the *onboarding* model), as the Android driver only has to be initialised once. This driver can be accessed by other sub-models via method in the AndroidBaseTestClass due to inheritance.

```
@BeforeExecution
public void Home() { driver = getDriverInstance(); }
```

Figure 24 before execution required for all models which do not contain the start element

Node and Edge behaviour implementation

The test-code was developed using Page Object Modelling (POM) design pattern. According to many practitioners POM, "*enhances test maintenance and reduces code duplication"* [23, 24]. The main logic behind POM is to, "*keep locators and test-cases separate from each other"*. Each page within both SUTs has its own object-oriented class, which stores the locators, which are used to execute the node and edge behaviour. Another advantage experienced during developing of the test code using POM, is whenever a failure occurs in the MBT test suite, the test engineer only has to apply the changes to the Page Object class to fix the issue.

For example, for the node `v_In_Home_Page_SHARED`, the Java test code is shown in Figure 27, the system used the `venueCheckInButton, checkSymptomsButton, readAdviceButton` (of type *id*) locators (as to verify that the current node was in fact the *home page*, by asserting that the elements where visible, with a 10 second timeout in place if the locator is not in view.

```
private final By venueCheckInButton = MobileBy.id("uk.nhs.covid19.production:id/optionVenueCheckIn");
private final By checkSymptomsButton = MobileBy.id("uk.nhs.covid19.production:id/optionReportSymptoms");
private final By readAdviceButton = MobileBy.id("uk.nhs.covid19.production:id/optionReadAdvice");
```

Figure 25 Example of locators used in HomePage class

These assertions are known as *Soft* assertions as opposed to the more common approach of *Hard* assertions which would halt the execution of the test suite if they failed by throwing an AssertException. As a means to incorporate failure tolerance, *Soft* was the preferred type in order to allow the test execution to progress to the next step despite the assertions failing. Although *Soft* assertions does not throw an exception, it collects the errors during the test and if you want to throw an exception, then you need to use *assertAll* method in the test suite as your last command.

```
public void checkElementVisible(By locator, int timeout) {
    SoftAssert softAssertion= new SoftAssert();
    softAssertion.assertTrue(isElementVisible(locator,timeout),  message: locator.toString() + " element visibility is.");
}
```

Figure 26 check Element Visibility via locator (Soft Assertion used)

During the development of the test code, a number of common Appium methods were developed for utility purposes which included *Android driver waits*, *scroll to find locator*, *check element is clickable* and *Android Debug Bridge (adb) commands* (used to control your device and get information from the device) to name a few. Through "modular programming" principles and applying *Software test-code engineering* (STCE) practices these methods where interchanged in the node / edge test cases as seen with `checkInHomePage` method in Figure 27.

```
@Override
public void v_In_Home_Page_SHARED() {
    HomePage homePage = new HomePage(driver);
    homePage.checkInHomePage();

}
```

And inside the *HomePage* class

```
public void checkInHomePage () {
    System.out.println("Check in Home Page");
    new WebDriverWaits(driver).checkElementVisible(venueCheckInButton,  timeout: 10);
    new WebDriverWaits(driver).checkElementVisible(checkSymptomsButton,  timeout: 10);
    new WebDriverWaits(driver).checkElementVisible(readAdviceButton,  timeout: 10);
}
```

Figure 27 Java Appium code implementing the behaviour of v_In_Home_Page_SHARED in Figure 16

Test-code development occurred incrementally, with regular running of test models as the Appium node /edge behavioural methods were developed to test the test suites and to make any necessary adjustments. A noted difference of MBT approach with a mobile application compared to a web application is the screen size. The automation developed requires scrolling, and in order to master the technical nuances of scrolling, the system used Android's UiScrollable (documentation) library to perform various scrolling actions. As you can see

below (in Figure 28) UiScrollable here is used to scroll the UI element (`resource id: linkTermsOfUse`) into view before clicking on it in order to perform the action for the edge `e_Click_Privacy_Notice` which is found in the About This App model.

```
public void scrollToFindElementBySelector(String selector){
    By locator = MobileBy.AndroidUIAutomator
            ( uiAutomatorText: "new UiScrollable(new UiSelector().scrollable(true).instance(0)).scrollIntoView(resourceId(\"" + selector +"\"))");
    driver.findElement(locator);
}
```

```
@Override
public void e_Click_Terms_Of_Use() {
    UIActionHelper uiActionHelper = new UIActionHelper(driver);
    AboutThisAppPage aboutThisAppPage = new AboutThisAppPage(driver);
    uiActionHelper.scrollToFindElementBySelector("uk.nhs.covid19.production:id/linkTermsOfUse");
    aboutThisAppPage.clickTermsOfUseLink();

}
```

Figure 28 usages of UiScrollable to find element by selector

As mentioned at the start of Section 4.1 - Test Models Implementation, UI inputs was one of the key implementations which was required for both SUTs. The use-cases for both apps came in the form of text input, for example enter test code, or the post code of your local district. This was fairly trivial to implement in Java using Appium. For example, the edge `e_Invalid_Postcode_District_Code` in Figure 29 showcases the test code which is using "Page Object" design pattern, is relatively short in terms of lines of code (LOC). The postcode text is located using the `resource-id`, then it cleared and an invalid postcode of '*33*' is entered. This edge is also an example of an *error checking* (node to itself) use-case within the MBT test suites.

```
@Override
public void e_Invalid_Postcode_District_Code() {
    EditPostcodePage editPostcodePage = new EditPostcodePage(driver);
    editPostcodePage.enterInvalidPostcode();
    editPostcodePage.clickContinuePostcode();


}
```

```
public void enterInvalidPostcode() {

    System.out.println("Enter Invalid Postcode");
    new WebDriverWaits(driver).waitForElementToBeClickable(postCodeEditTextButton, timeOutInSeconds: 10);
    driver.findElement(postCodeEditTextButton).clear();
    driver.findElement(postCodeEditTextButton).sendKeys( ...charSequences: "33");
}
```

Figure 29 Java Appium code implementing the behaviour for edge e_Invalid_Postcode_District_Code

Section 4.4 – Execution of MBT test suites

After the test models were designed, validated and the required test automation code was used to implement the nodes / edges' behaviour in Java using Appium was completed, it was then possible to start end-to-end runs of the MBT test suites on the two SUTs (NHS Covid-19 and Protect Scotland). As discussed in Model components / configuration section, GraphWalker supports wide range of configurations which can be applied before executing the MBT test suites.

Generator and Stop conditions implementation

When it came to deciding upon which generator and stop conditions to use, a number of factors had to be considered, (1) overall performance in terms of number of defects detected. (2) Test duration, it had to be a realistic amount of time, without comprising the integrity and accuracy of test execution. To begin with, the system was configurated using the *random generator* and with edge coverage goal=100% used as the *stop condition*. The documentation states that the Random generator, "*Navigates through the models in a completely random manner, also called random walk. This algorithm selects randomly an out-edge from a vertex and repeats the process for the next vertex."*

The logic behind this was the more random the path taken, the more potential there was to discover defects, however, the *random generator* proved to be extremely time consuming, and not practically viable for the project's scope. One possible solution could be to schedule "*test runs"* overnight using an automation solution such as Jenkins. However due to device capacity limitation (full list can be found in Section 2 - System Requirements and Specification), it was decided that another traversal algorithm that is offered by GraphWalker would be preferable.

According to GraphWalker there are a total of 4 different generator algorithms offered: *random, weighted random, quick random and A-star*. The next to be used was the *quick random* generator which uses Dijkstra's algorithm, to select the shortest path through each model. However, a noted drawback of *quick random* is that, when used in conjunction with *guards*, the algorithm can choose a path which is blocked by a *guard*, this limitation in GraphWalker API resulted in the system not using *quick random* for test teardown. Finally, the *weighted random* was implemented, which GraphWalker phrases as: "*the same as the random path generator, but will use the weight keyword when generating a path. The weight is assigned to*

edges only, and it represents the probability of an edge getting chosen." As the models for the two SUTs where large with an extensive onboarding process, *weighted random*, was better suited as it allowed to dramatically decrease the execution time without compromising the randomness of the path selected for the most part.

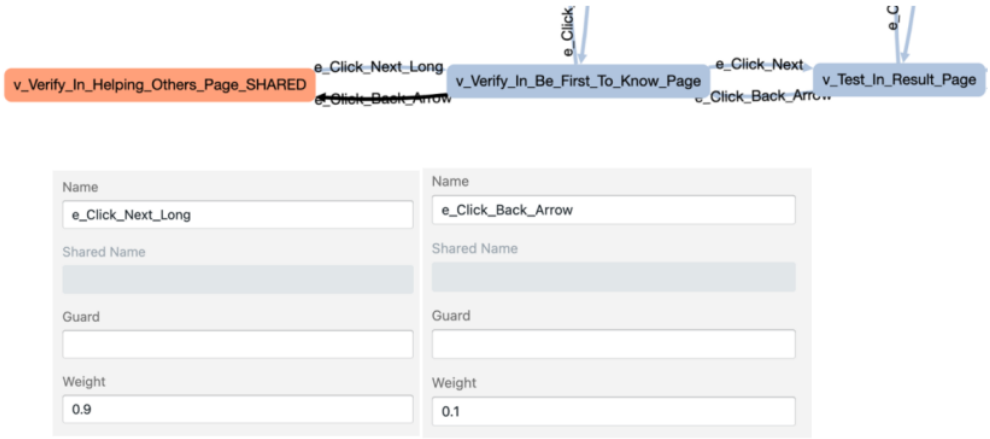


Figure 30 Weights applied to Onboarding Model of Protect Scotland SUT

After some initial experimentation described above, and with the two key factors considered, it was decided that *weighted random* generator would be used, with weights applied during setup wizards / on boarding process with 0.9:0.1 ratio in place on edges which takes you to the next stage as appose to edges take you back to the previous stage (as shown in Figure 30 above). This decision was based on observations when performing test runs with the *random* generator, a pattern emerged where a lot of time was spent looping back and forward between the initial sub-models, before eventually advancing '*deeper*' into the model. This strategy did not increase the perceived usefulness of the generator in terms of the test cases created and would benefit from applying *weights* to those loopback edges to reduce the time spent on the onboarding stages within both SUTs. As a result, the path chosen was 90% more likely to progress the next stage of onboarding process than to click the back button.

The system remained with the edge coverage goal=100% as the stopping condition, which allowed the MBT test suites to compressively test every single test-case and is generally seen as reasonable stopping condition in other industrial case studies. This left the final executable configuration in its GraphWalker notation as:

```
weighted_random(edge_coverage(100))
```

Test Executor and visualisation of test execution

The MBT test suites were implemented in a way to allow them to be executed via two different methods. The first is via Maven GraphWalker plugin, which requires the -Dgroups parameters to be set as shown in Section 4.2 – GraphWalker Java implementation, to indicate which SUT you want to run the MBT test suite for and GraphWalker will check for classes with the @GraphWalker annotation, which contains the corresponding *groups* name.

The second way to execute the MBT test suite, also starts a GraphWalker WebSocket server, which passes the executor as parameter in order to visualise live playback on GraphWalker Studio via a port configured in the proprieties file. The GraphWalker Test class uses JUnit 5 as means to kick of the test and GraphWalker Executor class to execute the MBT test suites. Below is an example of the NHS Covid-App test execution, with all the implemented GraphWalker models passed into the TestExecutor as parameters.

```
@Test
public void nhsAppAll() throws IOException {

    Executor executor = new TestExecutor(OnboardingModel.class,
            EnterTestCodeModel.class,
            SelectSymptomsModel.class,
            ManageMyDataModel.class,
            SettingsModel.class,
            qub.vg05.systemui.nhsmodels.AboutThisAppModel.class,
            qub.vg05.systemui.nhsmodels.HomeModel.class,
            RiskAreaModel.class,
            EditPostCodeModel.class,
            VenueCheckInModel.class,
            ReviewSymptomsModel.class,
            BookTestModel.class
    );
    WebSocketServer server = new WebSocketServer(webSocketPortNumber, executor.getMachine());
    server.start();

    setExcel(executor);
    result = executor.execute( b: true);
    errorControl();

}
```

Figure 31 NHS-Covid 19 Test execution with live Visualisation of Test progression

As well as live visualisation of test execution, both methods of execution include command line logs of the test as it happens, which includes custom messages of the action being performed and GraphWalker internal logs of each method which is being executed as shown in Figure 32, which includes: (1) execution of tracingActive guard on `e_Click_Contact_Tracing_Turn_OFF`, (2) execution of method `e_Click_Check_Symptoms` which clicks on the '*check symptoms button*', (3) execution of

`v_Verify_In_Select_Symptoms_Page_SHARED` to verify the state (i.e., the path has moved to the Select Symptoms page successfully)

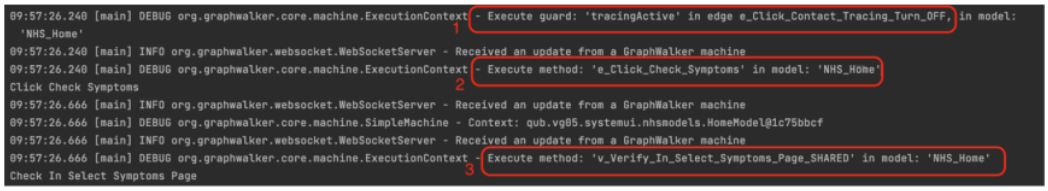


Figure 32 command line live logs of test execution

Without these live metrics, the tester has to wait until the end of MBT execution to see the outcomes, therefore it critical to have a real time view of how the test execution is progressing. GraphWalker studio also provides live updates for each sub-model, in terms of the number of steps taken (edges or nodes) and what the current fulfilment is within that sub-model. For example, in Figure 33, 24% of the elements (edge or nodes) have been visited in a total of 12 steps for the onboarding sub-model.

Figure 33 sub-model steps and fulfilment during live online MBT test suite execution

Section 5 – Testing

Due to the nature of this project being test-based, (MBT test suites), most of the testing occurred by running the test suites and checking whether the result was as expected. For instance, if a test execution failed due to an issue with model design, the model would be further exanimated by live visualisation of the test execution using GraphWalker studio to identify the issue. This allowed for bugs to be quickly identified and a fix implemented. Furthermore, each MBT test run generates GraphWalker reports which includes details of the test run for each sub-model and reports any errors which have occurred. This could potentially be an error with the test-code opposed to a real fault detected with the SUT. As shown in Figure 34, the error occurred within the `OnboardingModel`, due to a typo with the `resource-id` (locator) in the test-code. The system also benefited from using "Page Object" principles, which meant the code change only had to be applied to one class.

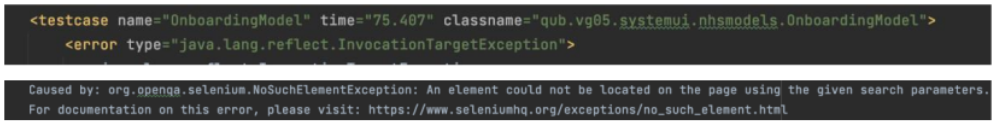


Figure 34 GraphWalker-generated report output used for test debugging

An enhanced log analysis Excel report is generated during the execution of MBT test suites, with metrics and statistics breaking down each MBT test execution. The enhanced reporting engine is integrated with GraphWalker tool and was based off the test log implementation in the open-source Model-based Testing (MBT) artifacts from MBT of Testinium (https://github.com/vgarousi/MBTofTestinium).

An example of what MBT test suite testing and the steps taken during the test path can been seen in Figure 35. The Excel log output, which is automatically generated after each test run, extends upon the reporting provided by GraphWalker and includes three separate sheets: (1) *TestPathDuration*, (2) *ModelEntityVisitCount* and (3) *ModelStats*. The *TestPathDuration* as shown in Figure 35, logs every step in the test path, with the time spend on each step (*shared* nodes are highlighted in grey). The "Model visit sequence number" column shows how many times the test path has visited each sub-model as of the current step in the execution. The "Entity visit sequence number" column shows the sequence of visiting a given node or edge within the MBT test suite. As you can see from example in Figure 35 the `v_Verify_Home_Page_SHARED` had been visited 691 times, before the last edge was traversed, fulfilling the stop condition of 100% edge coverage at 16.04PM (with the test commencing at the start node `v_Start` at 13.15pm).

The overall test result is included as the final column, with a failure column included for each step, this is used to reference which step (node or edge) the failure occurred on. The backlog of previous steps leading up to a failure, proved beneficial in order to understand what might have caused the failure and allowed for failure scenarios to be reproduced by tracing back to the model and the path taken.

	Test Model Name	Model visit seq num	Steps (via nodes and edges)	Entity visit seq num	Date	Time	DURATION for each step (seconds)	Failure?
2	Age_Confirmation	1	v_Start	1	21/04/2021	13:15:38.789	00:00.043	
3			e_Go_To_Age_Check_Page	1	21/04/2021	13:15:38.832	00:00.001	
4			v_Verify_In_Age_Page_SHARED	1	21/04/2021	13:15:38.833	00:03.843	
5	Settings	1	v_Verify_In_Age_Page_SHARED	1	21/04/2021	13:15:42.676	00:00.552	
6	Age_Confirmation	2	v_Verify_In_Age_Page_SHARED	2	21/04/2021	13:15:43.228	00:00.089	
7			e_Click_Am_12_15	1	21/04/2021	13:15:43.317	00:00.716	
8			v_12_15_Age_Confirmation_Message	1	21/04/2021	13:15:44.033	00:00.624	
9			e_Click_To_Confirm_Am_12_15	1	21/04/2021	13:15:44.657	00:01.767	
10			v_Verify_In_Parent_Optional_Check_Page	1	21/04/2021	13:15:46.424	00:00.063	
11			e_Click_Back_Arrow_VIPOCP8_VIAPS1	1	21/04/2021	13:15:46.487	00:00.792	
12			v_Verify_In_Age_Page_SHARED	3	21/04/2021	13:15:47.279	00:01.120	
13	Settings	2	v_Verify_In_Age_Page_SHARED	2	21/04/2021	13:15:48.399	00:00.560	
14	Age_Confirmation	3	v_Verify_In_Age_Page_SHARED	4	21/04/2021	13:15:48.959	00:00.086	
9956	Home_Tour	46	v_Verify_In_Settings_Home_Tour_Page_SHARED	69	21/04/2021	16:03:58.615	00:00.558	
9957			e_Click_Close_VISHTPS5_VIAAP3	8	21/04/2021	16:03:59.173	00:00.674	
9958			v_Verify_In_About_App_Page	31	21/04/2021	16:03:59.847	00:00.633	
9959			e_Click_Next_VIAAP3_VIACP4	20	21/04/2021	16:04:00.480	00:00.672	
9960			v_Verify_In_Add_Code_Page	35	21/04/2021	16:04:01.152	00:00.936	
9961			e_Click_Next_VIACP4_VIHP6	18	21/04/2021	16:04:02.088	00:00.671	
9962			v_Verify_In_Home_Page	18	21/04/2021	16:04:02.759	00:00.000	
9963			e_Go_To_Inactive_Home	9	21/04/2021	16:04:02.759	00:00.001	
9964			v_Verify_In_Home_Page_Inactive_SHARED	9	21/04/2021	16:04:02.760	00:02.298	
9965	Home	228	v_Verify_In_Home_Page_Inactive_SHARED	9	21/04/2021	16:04:05.058	00:00.031	
9966			e_Click_Turn_On	3	21/04/2021	16:04:05.089	00:04.669	
9967			v_Verify_In_Turn_On_Covid_Notifications_Page	3	21/04/2021	16:04:09.758	00:00.295	
9968			e_Click_Cancel	1	21/04/2021	16:04:10.053	00:00.435	
9969			v_Verify_In_Home_Page_Inactive_SHARED	10	21/04/2021	16:04:10.488	00:10.021	
9970			e_No_Action_Pop_Finished	1	21/04/2021	16:04:20.509	00:00.001	
9971			v_Verify_Home_Page	691	21/04/2021	16:04:20.510	00:02.044	
9972	TEST_RESULT				21/04/2021	16:04:22.554		Successful

Figure 35 Partial snapshot of output test log, showcasing test paths and entity and model visit sequence sum, which is generated automatically for each MBT test suite execution

The *ModelEntityVisitCount* sheet includes the aggregated visit count for each model, as well as each entity (node and edges) and the total time spent in each model entities. This allows the test engineer to establish an overall view of the usages rate for each node and edge, to understand which edges / nodes are the most traversed and subsequently which are the least traversed. For both SUTs, the home page node was by far the most traversed node (1019 NHS Covid-19 and 619 Protect Scotland) recorded on a recent test execution (21/04/2021). As you can see from Figure 36 a recent MBT execution of Protect Scotland test suite took a total 9970 steps, over the course 2 hours and 48 minutes before reaching the required stop condition of 100% edge coverage.

The final sheet generated is *ModelStats* which provides a high-level view of each sub-model, this makes up the overall SUT. An example of a recent ModelStats generated for the NHS Covid-19 SUT can been seen in Figure 37.

1	11	11	183				
2	**Model #**	**Test Model Name**	**Entity (nodes and edges)**	**Visit count of entity**	**Total visit count of model entities**	**Total time on entity**	**Total time on model**
3	1/11	Age_Confirmation	v_Verify_In_Age_Page_SHARED	171		00:01:13.606	
4			v_Start	1		00:00:00.043	
5			v_12_15_Age_Confirmation_Message	42		00:00:23.200	
6			v_Verify_Secondary_School_Check	91		00:01:12.782	
7			v_Over_16_Age_Confirmation_Message	36		00:00:18.487	
8			v_Verify_Secondary_School_Warning_Message	35		00:00:22.225	
9			v_Verify_In_Parent_Required_Check_Page	25		00:00:15.686	
10			v_Verify_In_Parent_Optional_Check_Page	25		00:00:08.121	
11			v_Verify_In_Location_Page_SHARED	93		00:00:43.674	
12			e_Go_To_Age_Check_Page	1		00:00:00.001	
...							
77			e_Click_Close_Chrome	83		00:00:51.348	
78			e_Click_Book_A_Test	38		00:01:02.545	
79			e_Click_Add_Test_Code	50		00:00:12.978	
80			e_Click_Back_Arrow	23		00:00:20.440	
81			e_Click_Close	27	610	00:00:13.471	00:08:39.558
82	11/11	Enter_Test_Code	v_In_Enter_Test_Code_Page_SHARED	69		00:00:33.222	
83			v_Invalid_Test_Code_Message	46		00:00:33.037	
84			e_Enter_Invalid_Test_Code	46		00:00:36.496	
85			e_Clear_Text_Box	46	207	00:00:08.523	00:01:51.278
86		**Total**			**9970**		**02:48:43.765**

Figure 36 Example of Protect Scotland Test execution log (Aggregated Model entity visit count and total steps and duration of test run)

	A	B	C	D	E	F	G
1	**Model #**	**Test Model Name**	**# of Nodes**	**# of Edges**	**Total # of entities**	**Total time on model**	**SUM exec of entities**
2	1/12	NHS_Onboarding	12	20	32	00:03:31.485	1083
3	2/12	NHS_Select_Symptoms	9	18	27	00:05:16.746	1500
4	3/12	NHS_Enter_Test_Code	3	4	7	00:03:59.143	1120
5	4/12	NHS_Venue_Check_IN	6	8	14	00:04:52.306	1305
6	5/12	NHS_Home	13	25	38	00:15:57.581	2501
7	6/12	NHS_Review_Symptoms	11	23	34	00:03:19.584	490
8	7/12	NHS_Risk_Area	2	2	4	00:03:52.716	286
9	8/12	NHS_Settings	3	15	18	00:10:18.288	709
10	9/12	NHS_Book_Test	2	4	6	00:02:38.757	235
11	10/12	NHS_About_This_App	3	9	12	00:05:06.807	211
12	11/12	NHS_Edit_Postcode	4	6	10	00:00:19.682	38
13	12/12	NHS_Manage_My_Data	3	4	7	00:00:09.644	39

Figure 37 Overview of Model stats for each sub-model for test execution of MBT test suite for NHS Covid-19

Another example of testing occurred on February 18th, 2021 after a system update of the test device (Samsung Galaxy S10) from Android 10 to Android 11. This update changed the system level OS permission controller elements (camera permissions menu), which caused the test suite to fail, as the locators had changed with the new version of Android. This was to be expected, and as a result of this testing, the test suites were updated to support both Android 10 and 11 versions.

Section 6 - System Evaluation

To assess the MBT test suites developed, the system will be compared against the requirements outlined in Section 2.3 – Requirements, as well as evaluating the benefits of the MBT approach used, any limitations observed and possible improvements to the current implementation.

Section 6.1 - Benefits of MBT approach of Android Application

Throughout the course of this project, a number of benefits of MBT testing approach has been observed:

Increased overall test coverage and effectiveness

When comparing MBT testing approach to other testing methods, it achieves an increased test coverage, with a high fault detection to code size ratio. Using MBT has increased testing thoroughness, in terms of generating test cases representing realistic user behaviour, that may have been missed via other testing methods. The number of test-cases and test steps generated are exponentially greater than other standard black-boxing testing approaches, thus also ensuring the test suites are cost-effective.

Both MBT test suites typically take over 2 hours with an average of 8000 test steps generated randomly. This turned out to be an advantageous strategy at producing sequences in the model to cover UI elements in ways that may not have been considered. The MBT test suites developed would be ideal to for "regression testing" (running the MBT suite every time a change has been made to the system) at no extra cost. Furthermore, research on model-based regression testing concluded, *"regression testing can be supported and enhanced by model–based testing"* [25] .

Improved test case design

The test-case design for the UI elements is usually a manual process, this can lead to test-cases being ad-hoc. MBT allows for improved test-case design due to systematic coverage using test models and removal of redundancies which effects the overall quality of the test cases. Before implementing MBT on SUTs, some manual test cases were designed for Protect Scotland application, for the Onboarding pages.

When compared with those generated from MBT approach using GraphWalker it is clear that many test paths were overlooked. With GraphWalker transversal algorithm *random* the number of test steps generated can vary greatly from one test run to the next. The comparison of various MBT approaches and the initial manual approach can be seen in Table 2. Note, to account for the variations in the test paths generated by GraphWalker (*random* or *weighted random* generators), an average was taken for the "number of test steps generated" and the "edge coverage" runs for each technique used. The results show that MBT generates a higher number of test steps using a range of stop conditions compared to manual approach and full edge coverage was achieved by those that applied *edge_coverage (100)* stop condition, with all others achieving partial edge coverage. The weighted random was configured with selective weights applied to edges on onboarding models, to reduce loopback cycles. It is from these findings that MBT has resulted in improvements in test-case design practices, allowing for optimal edge coverage as well as covering each requirement multiply times more, than manually designed test cases.

Table 2 Comparison between MBT generated test cases using a range of stopping conditions provided by GraphWalker (All comparison below were on Protect Scotland test suite)

Test Case Design Technique	**Stopping Conditions**	**Transversal Algorithm**	**Number of test steps generated**	**Edge Coverage %**
Manual	until the end of manually designed test cases	sequential	41	32
MBT (1)	edge_coverage (100)	random	16183	100
MBT (2)	edge_coverage (100)	weighted random	9970	100
MBT (3)	vertex_coverage (100)	random	4967	97
MBT (4)	length (1000)	random	1763	90

Decrease in the number of test smells

The number of test smells compared to typical blacking-box testing has been greatly reduced through the MBT approach. Typical test smells occur in test suites as a result of bad design or test patterns. Using UI test models to generate test-cases has enabled high maintainability of test code and decreased the number of test bugs, as the test-cases are less brittle and are simply

to implement. The test-cases produced by MBT Approach used "*good test practice*" applying Software Test-Code Engineering (STCE) principles [9], in modularising the test-code by using verification rules under a function and calling it from the node or edge testcase, hence avoiding *eager* test cases. A research paper on "improvements of test-code" [18], concluded, *"an eager test is a test that verifies too much functionality. It is mostly, but not necessarily, a test with a large amount of statements and assertions."*

Requires little maintenance

Little maintenance on test suites has been required once the models have been designed and the test cases implemented. Although the setting up of MBT testing approach has been complex and required learned skills to design the test models following best practices, once finished it requires little maintenance and the benefits gained are well worth the learning curve that it entails.

The enhanced test log output has allowed for the root source of failing tests to be quickly identified within the test models. Additionally, any changes in the SUTs can also be reflected in the model (the node and edge behaviour implementation which are already in place are automatically reused due to the modularity approach used when developing them). Furthermore, any change in the behaviour of performing an edge (transition action) or node (assertions on a GUI element), is isolated to a single node or edge implementation for common edges such as `e_Click_Close` which is reused 5 times for home page model (Protect Scotland) 3 of which are highlighted in Figure 38.

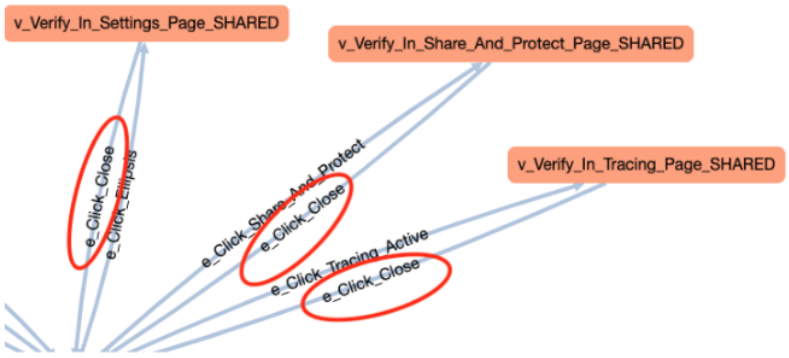


Figure 38 Example of common edges which reused with Protect Scotland - home page model

Intangible benefits

This is based on the opinion of the developer involved in this project. When developing the MBT test suites, the phases of work required was more "engaging and interesting" compared to previous testing experience with other forms of black-box testing. Although these benefits are difficult to measure, it is an important footnote when making a decision about which testing approach to use in an industrial context.

Section 6.2 - Limitations / weaknesses

The following sections includes a list of limiting factors / shortcomings, which as a result has impacted the project and its validity.

Test device capacity

A benefit of MBT test suites is that they can be executed on multiple test devices simultaneously and each test path generated will be unique when using *weighted random* traversal algorithm. However, for this project there was only one test device available, which proved to be a limiting factor in terms of the system's ability to perform endurance testing on the SUTs.

Lack of MBT coverage tool usage

The test suites do not include an MBT testing coverage tool. This would have provided a greater ability to assess the effectiveness of the MBT approach implemented in terms of how well the code is being tested, with features such as *"cumulative"* coverage. A possible improvement could be to use a tool such as MBTCover (github.com/vgarousi/MBTCover).

Android implementation only

The MBT testing suite works on the Android platform exclusively, meaning the Protect Scotland and NHS Covid-19 on iOS has not been covered within the scope of this project. Theoretically the abstract models would still work with iOS, but the test-code would have to be implemented in order to connect to an iPhone. Appium could still be used, but iOS requires XCUITest Driver in order to communicate and stimulate user actions on Apple devices, which could possibly be included with future works.

Limited testing of Graph traversal algorithms (offered by GraphWalker)

Not all graph traversal algorithms where comprehensively tested when deciding which would be the most appropriate for the models designed for both SUTs. The system ended up using the

weighted random options after limited testing (of *weighted random and random)* due to time constraints.

Device Compatibility

The test code implementation using the Appium framework, has only been tested on Android 10/11 device. This restriction could lead to compatibility issues with versions outside this remit. If resources allowed, the MBT test suites would have been executed on multiple devices, all with different configurations, to increase the confidence in the implementation of MBT approach on the SUTs.

Lack of direct comparison to assess benefits

The SUTs (NHS Covid-19 and Protect Scotland applications) has no record or access to other UI testing approaches applied to them (other than basic manual testing of Protect Scotland which was performed at the start of the project); therefore, no direct comparison was able to be made with MBT approach applied by the system developed in this project.

Section 6.3 - Conclusion and Future work

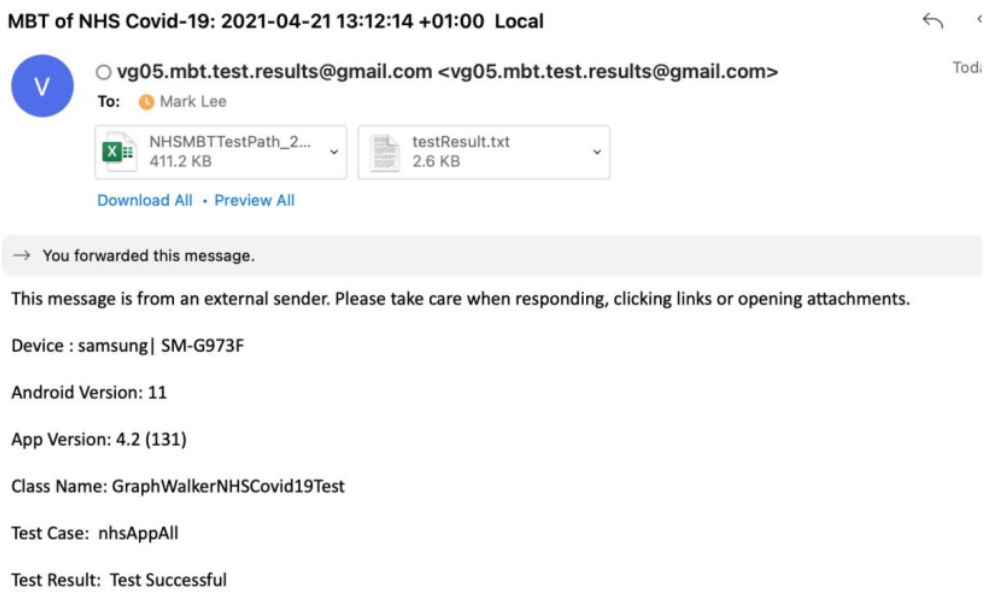
In this report a test system has been presented which applies MBT approach to testing Covid-19 Android applications, executed on a real device (Samsung Galaxy S10). The solution approach implemented has produced a positive outcome in terms of testing the functionality of the two SUTs and validated that both SUTs were working as expected, however there is still room for improvement through future iterations of the system. In terms of MBT approach within the *mobile application* domain, a number of benefits have been observed (see Section 6.1 - Benefits of MBT approach of Android Application), which could prove helpful with future work within the domain.

Although the MBT test suites have fulfilled all of their requirements, there are still areas that could be improved upon which include, but are not limited to: (1) iOS implementation – by using a high abstraction level the models would be reusable on iOS platform (test code implementation required using XCode), (2) inclusion of MBT coverage tool, and (3) implement other methods of black box testing in order to gain data points to compare MBT approach with. This would allow for a quantitative assessment of the benefit of MBT approach to testing.

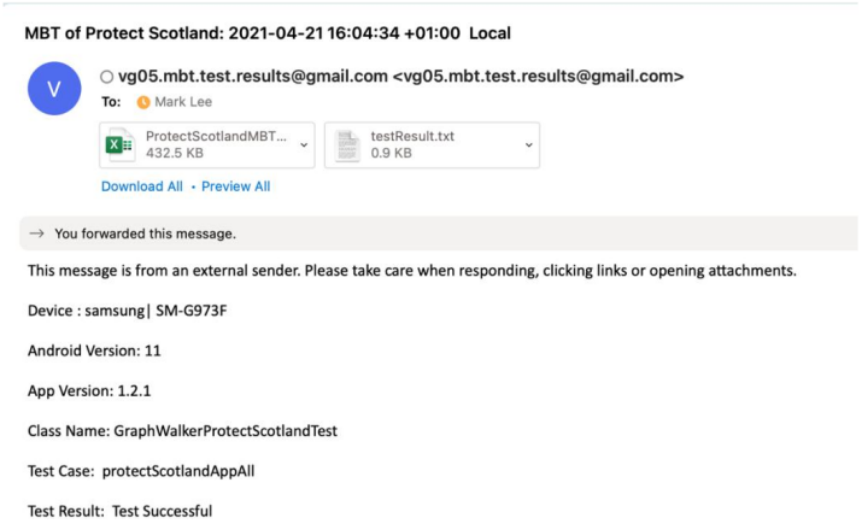
Section 7 – Appendix

Section 7.1 - Email generated after execution of MBT test suites

NHS Covid-19

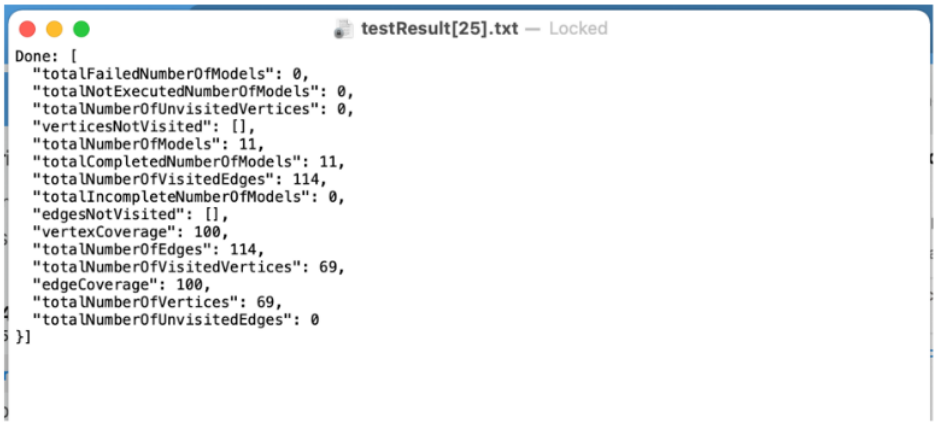


Protect Scotland

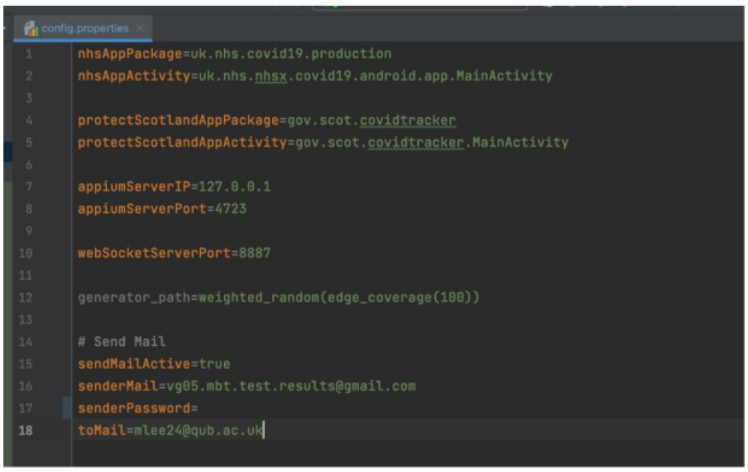


Section 7.2 - GraphWalker basic test result output

Protect Scotland example



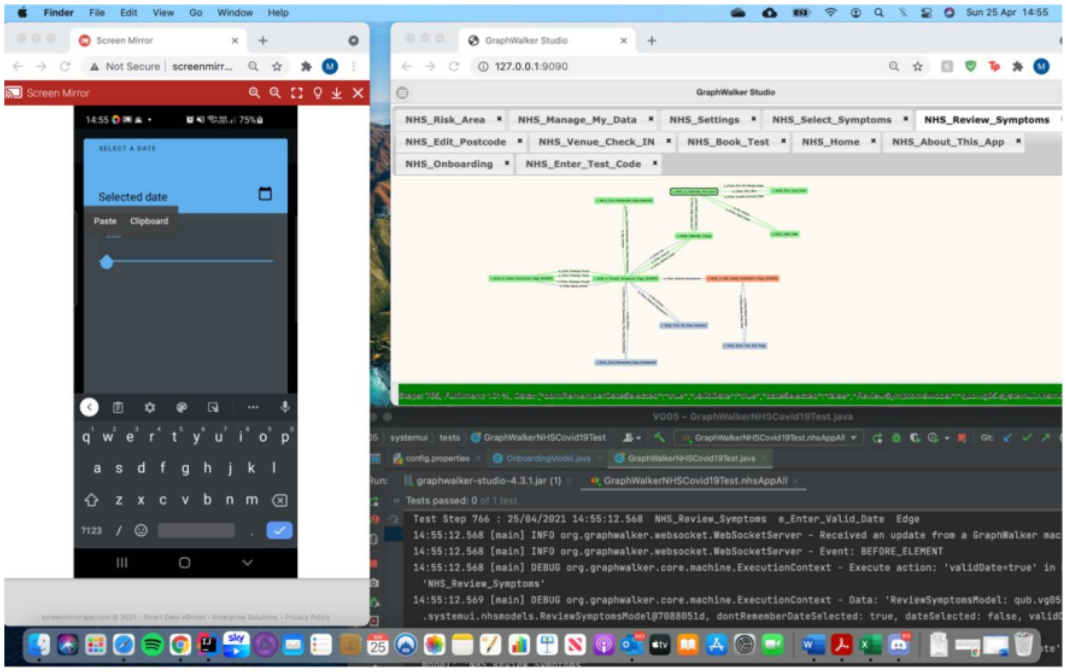
Section 7.3 – Configuration properties file



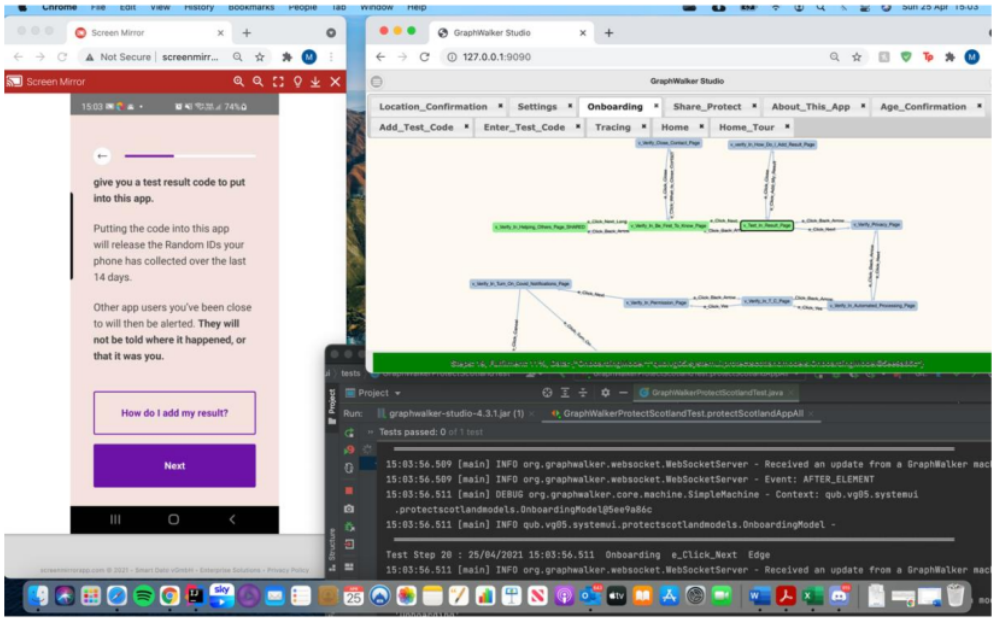
Section 7.4 – MBT GraphWalker Execution

MBT execution of test suites, which are connected GraphWalker studio to visualise test progression on the test models.

NHS Covid-19



Protect Scotland



Reference

The source code for this project can be found at the following EECS Gitlab repository

- GraphWalker MBT of NHS Covid-19 and Protect Scotland Android applications: https://gitlab2.eeecs.qub.ac.uk/401516151/vg05-development-of-automated-test-suites-to-test-covid-contact-tracing-apps

[1] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Transactions on Reliability,* vol. 68, no. 1, pp. 45-66, 2018.

[2] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann and D. Lo, "Understanding the Test Automation Culture of App Developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Graz, Austria, 2015.

[3] H. Wang, H. Li, L. Li, Y. Guo and G. Xu, "Why are Android Apps Removed From Google Play? A Large-Scale Empirical Study," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, Gothenburg, 2018.

[4] MIT Technology Review, "Covid Tracing Tracker," 2021. [Online]. Available: https://docs.google.com/spreadsheets/d/1ATalASO8KtZMx__zJREoOvFh0nmB-sAqJ1-CjVRSCOw/edit#gid=0. [Accessed 2 Feburary 2021].

[5] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink and A. Bacchelli, "On the Relation of Test Smells to Software Code Quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, 2018.

[6] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering* , vol. 20, no. 4, pp. 1052-1094, 2015.

[7] Y. L. Arnatovicha and L. Wang, "A Systematic Literature Review of Automated Techniques for Functional GUI Testing of Mobile Applications," A Systematic Literature Review of Automated Techniques for Functional GUI Testing of Mobile Applications, Singapore, 2018.

[8] V. Garousi, A. B. Keleş, Y. Balaman, Z. Ö. Güler and A. Arcuri, "Model-based testing in practice: An experience report from the web applications domain," 2020.

[9] V. Garousi, Y. Amannejad and A. B. Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology,* vol. 58, pp. 123-147 , 2015.

[10] T. Takala, M. Katara and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, Berlin, 2011.

[11] A. R. Espada, M. d. M. Gallardo, A. Salmerón and P. Merino, "Performance Analysis of Spotify® for Android with Model-Based Testing," *Mobile Information Systems,* 2017.

[12] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu and Z. Su, "Practical GUI Testing of Android Applications Via Model Abstraction and Refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, 2019.

[13] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan and I. Porres, “Model-based testing using UML activity diagrams: A systematic mapping study,” *Elsevier Computer Science Review,* vol. 33, pp. 98-112, 2019.

[14] inflectra.com, “Model-Based UI Testing,” 8 Feburary 2017. [Online]. Available: https://www.inflectra.com/support/knowledgebase/kb284.aspx. [Accessed 15 12 2020].

[15] M. Kropp and P. Morales, “Automated GUI Testing on the Android Platform,” in *22nd IFIP International Conference on Testing Software and Systems: Short Papers*, Quebec, 2010.

[16] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta and G. Sagardui, “Model-Based Testing in Practice: An Industrial Case Study using GraphWalker,” in *ISEC 2021*, Bhubaneswar, 2021.

[17] V. Garousi, A. B. Keleş, Y. Balaman, Z. Ö. Güler and A. Arcuri, “Model-based testing in practice: An experience report from the web applications domain,” 5 April 2021. [Online]. Available: https://arxiv.org/abs/2104.02152. [Accessed 15 April 2021].

[18] V. Garousi and M. Felderer, “Developing, Verifying, and Maintaining High-Quality Automated Test Scripts,” *IEEE Software,* vol. 33, no. 3, pp. 68-75, 2016.

[19] M. Barth and A. Fay, “Automated generation of simulation models for control code tests,” *Control Engineering Practice,* vol. 21, no. 2, pp. 218-230, 2013.

[20] S. Rösch, S. Ulewicz, J. Provost and B. Vogel-Heuser, “Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains—Current Challenges and Research Gaps,” *Journal of Software Engineering and Applications,* vol. 8, no. 9, 2015.

[21] A. C. D. S. L. Stefan Karlsson, “Model-based Automated Testing of Mobile Applications: An Industrial Case Study,” 20 Augest 2020. [Online]. Available: https://arxiv.org/abs/2008.08859.

[22] V. Garousi, “Test automation with the Gauge framework: Experience and best practices,” ICCSA, 2020.

[23] Brightest, “An introduction to POM (Page Object Model),” 12 Augest 2020. [Online]. Available: https://www.brightest.be/business/an-introduction-to-pom-page-object-model/. [Accessed 01 March 2021].

[24] V. Garousi and E. Yldrm, “Introducing automated GUI testing and observing its benefits: an industrial case study in the context of law-practice management software,” in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018.

[25] P. Zech, P. Kalb, M. Felderer, C. Atkinson and R. Breu, “Model–based regression testing by OCL,” *International Journal on Software Tools for Technology Transfer,* vol. 1, no. 19, 2017.

[26] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu and Z. Su, “Guided, Stochastic Model-Based GUI Testing of Android Apps,” in *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, New York, 2017.