

Developing fault tolerance features for an open-source Model-based Testing (MBT) tool

A dissertation submitted in partial fulfilment of
the requirements for the degree of
BACHELOR OF *SCIENCE* in Computer Science
in
The Queen's University of Belfast
by
'Glen O'Donovan'
'10/05/2022'

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER
SCIENCE**

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name:	Glen O'Donovan	Student Number:	40233306
Project Title:	Developing fault tolerance features for an open-source Model-based Testing (MBT) tool		
Supervisor:	Dr Vahid Garousi		

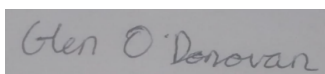
Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at <http://www.qub.ac.uk/cite2write/introduction5.html> and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)



Glen O'Donovan

10/05/2022

Acknowledgements

I would like to thank Dr Vahid Garousi as my project supervisor for providing me guidance during the course of this project, as well as access to the Testinium test suite. I would also like to say my thanks to Testinium for allowing me to practice and test my heuristics on their website. As well as thanks to Yunus Balaman who showed me how to set up the Testinium test suite. Finally thanks to Kristian Karl who developed GraphWalker and has made this software open-source [13].

Abstract

MBT is a test approach, it often suffers from a limitation: when an assertion fails in the test models, execution of the test models stops altogether, and does not continue the execution from other parts of the model. Thus the aim is to incorporate certain “fault tolerance” features in the selected MBT tool.

The two heuristics in mind to implement is: (1) when the assertion in a model node fails, go back from the node to the previous node and go the failed node again; (2) when the assertion in a model node fails, walk back from the node to the previous node and continue the MBT execution from there to other nodes.

The system these heuristics were tested on was Testinium. The results were largely positive in that the two heuristics were successfully implemented and tested. While I believe the heuristics are a good addition to anyone using the MBT tool GraphWalker, of course there is still room for future development to improve and expand on what has been done in this project.

Contents Page

Acknowledgements	3
Abstract	3
Contents Page	4
Introduction and Problem Area	5
Background:	5
Problem area:	6
System Requirements and Specification	7
System Requirements	9
Design	11
Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again	13
Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)	15
Implementation	17
Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again	19
Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)	20
Summary of changed code in and added code to GraphWalker	23
Testing	25
Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again	27
Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)	31
System Evaluation and Experimental Results	35
Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again	35
Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)	36
Possible Future Improvements	37
References	39
Appendices	41
User guide on how to switch from one heuristic to another:	41

Introduction and Problem Area

Background:

Systematic and proper testing of software systems is a tedious and costly task, as are the costs of software defects due to insufficient testing. Creating tests manually is error-prone and time-consuming because it requires a multitude of different test scenarios. Researchers have proposed several methods for automating both the creation and execution of tests.

Test automation [14] has been used for several decades in order to improve the effectiveness and efficiency of testing. While most software testers use automation for the test execution phase, test automation is “not just for test execution” [2], i.e., it can be used in other test activities such as test-case design.

Model-based testing (MBT) [3] is an established black-box testing approach for generation of test cases. In MBT, specific types of models, often called test models, are developed or are reused from earlier software lifecycle phases (e.g., requirements or design) for generation of test cases. Within each of these models, there are several nodes which is where you would have assertions in your code. These assertions verify that an API call returns the correct values, that a button click actually did close a dialog, or that when the timeout should have occurred, the System Under Test triggered the expected event. Between these nodes are a number of edges, these edges perform actions that transition you from one node to another. This action could be an API call, a button click, a timeout, etc.

MBT has been around for at least 50 years now. An IBM technical report [4], published in 1970, is often referred to as one of the first known reported applications of MBT. In this study GraphWalker will be the MBT tool used. GraphWalker [5] is a popular open-source MBT tool, which can create, edit, and execute the test models created.

Problem area:

From a model, GraphWalker will continuously generate paths as it traverses through said model. A model has a start node, and a generator which rules how the paths are generated, and associated stop condition which tells GraphWalker when to stop generating the path [8]. While MBT is a powerful test approach, it often suffers from a limitation: when an assertion fails in any one of the nodes in the test models, execution of the test models stops altogether, and does not continue the execution from other parts of the model. Thus, it is important to incorporate certain “fault tolerance” features in the selected MBT tool. In fact, the developer behind GraphWalker has included a “placeholder” for various fault tolerance heuristics in the code-base.

With the two heuristics mentioned previously in the Abstract, they help solve some problems MBT has. For example, sometimes tests fail and are non deterministic (called “flaky” tests) and the assertion may pass the second time executing it. This study will show the effectiveness of the implementation of two fault tolerant features.

The web application under test to assess and validate the fault tolerance heuristics, developed in GraphWalker is Testinium. Testinium has pragmatically used MBT, since January 2019, to improve the company’s test-automation practices [9]. Therefore Testinium will be an adequate System Under Test in order to evaluate how well the new fault tolerant features work due to there already being an open-source MBT test suite available to use.

System Requirements and Specification

As previously stated in the background section, GraphWalker will be the MBT tool used in this study. For reasons such as that it is open source and that the developer behind GraphWalker has included a “placeholder” for various fault tolerance heuristics in the code-base. Now that we have identified the main tool that we are going to be using, let’s look at how we can use this tool in order to begin to provide a solution to the issues that we have previously identified.

At a high-level overview, we can say that our solution goal is to be able to perform GraphWalker tests against a system with either defects in the SUT in order to fail an assertion or mutate the assertion in our code to induce a failure in the running of the model. Once failed, the MBT tool should go back to the previous node and attempt to cross the edge and retry the assertion in the failed node. Or as the second heuristic, the tool should walk back from the node to the previous node and continue the MBT execution from there to other nodes(also making sure to flag the failed nodes as red, and not to visit them again – like a “Black list”)

The system-under-test (SUT) will be testinium [6], which was mentioned previously has used MBT before and therefore we have its MBT suite already available open-source [11]. This means that we can immediately start developing fault tolerance features on our MBT tool without having to design and develop the MBT test models from scratch for the SUT.

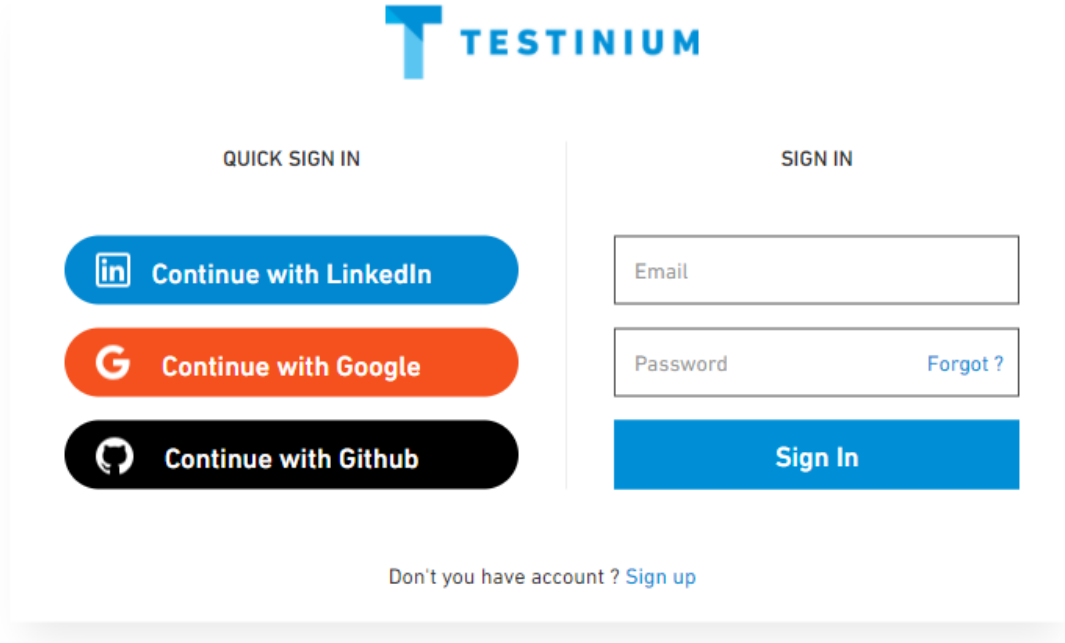


Figure 1 - Testinium SUT Login Page

This graphwalker solution tests the SUT using selenium, [7] which is an extension to emulate user interaction with browsers. The solution would be so, that if any of these nodes fail their assertion, that after trying it twice the MBT test suit would halt the execution of the test or try the node only once and then label that node as failed and to not visit that node again while continuing to test the SUT.

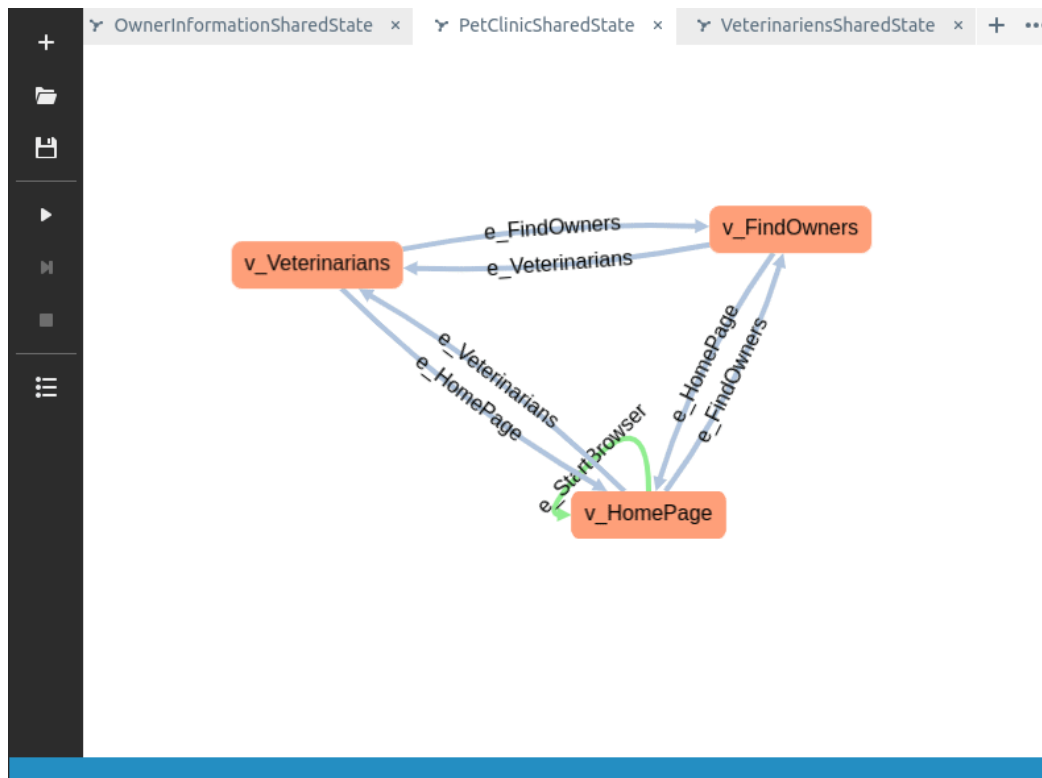


Figure 2 - Example Gif of Graphwalker test execution [12]

The GraphWalker player will be what is used to visualise the tests progress on a single web page. It allows test engineers to see the tests progress through the changing of colours on the graphwalker models such as in Figure 2.

System Requirements

Functional Requirements

- Depending on which of the two heuristics has been chosen on a given test run, the enhanced GraphWalker will try all nodes twice before halting the execution of the test or place them on the black list after one failed attempt and continue the test.
- The MBT will continue to run the MBT model until the stop condition is fulfilled, even when met with assertion fails if it is using heuristic 2.

Non-functional Requirements

- Clear and consistent developer documentation in the form of comments in source code and this paper.
- Functional and feasible system. Extension to GraphWalker implementation that provides fault tolerance features.

Software Environment

- GraphWalker CLI (Running GraphWalker)
- GraphWalker Studio (Developing models)
- Java
- Maven
- Selenium
- Git with GitLab
- Chromium DevTools

Design

Graphwalker Player has colour codes to represent what state the node or edge is in or has been in. Nodes and edges in light blue colours, are not yet visited (or executed by the test). When a node or edge is visited, it will be coloured light green. An edge or node currently being executed by the test will be highlighted with a black border. Nodes that have been visited/executed by the test and have failed we be coloured red.

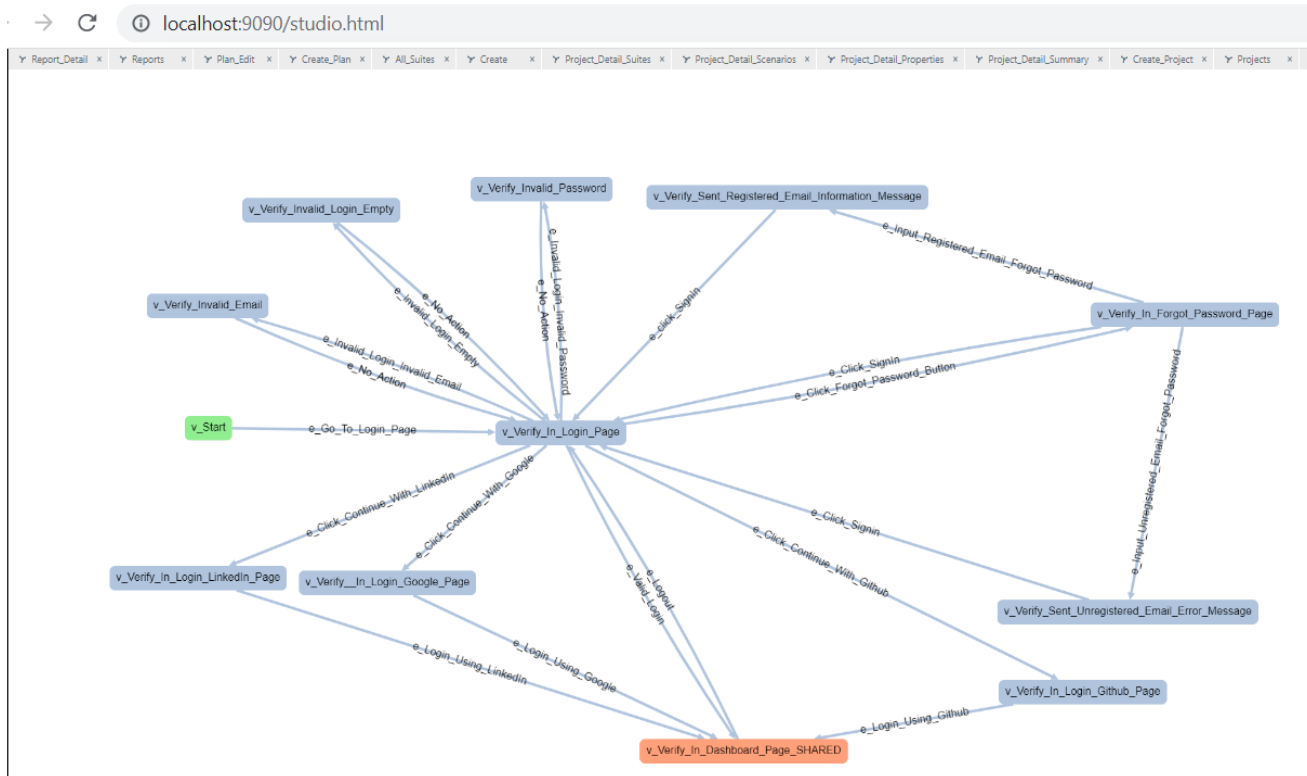


Figure 3 - “Login” MBT model of the SUT: Testinium

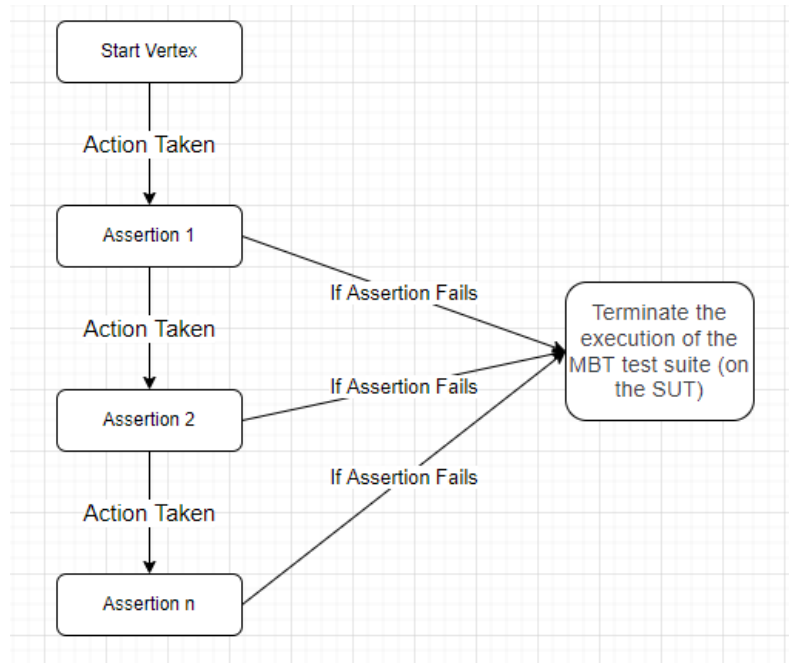


Figure 4 - GraphWalker with no fault tolerance Features

The way Graphwalker currently transverses is through a series or singular Path generator together with stop condition/s [8], which will decide what strategy is used when generating a path through a model, and when to stop generating that path. An example of a few generators are as follows:

- “random” where the MBT test suite navigates through the model in a completely random manner until the stop condition is fulfilled.
- “weighted_random” acts in the same manner as the previous generator but this time uses the weight keyword when generating a path. The weight is assigned to edges only, and it represents the probability of an edge getting chosen.
- “a_star” where the MBT test suite navigates through the model to a specific node or edge after generating the shortest path to that node or edge.

If an assertion fails, the program will stop as seen in Figure 4. The development and then implementation of the two fault tolerant heuristics is that if any of these assertions fails, the Path generators will continue to generate paths.

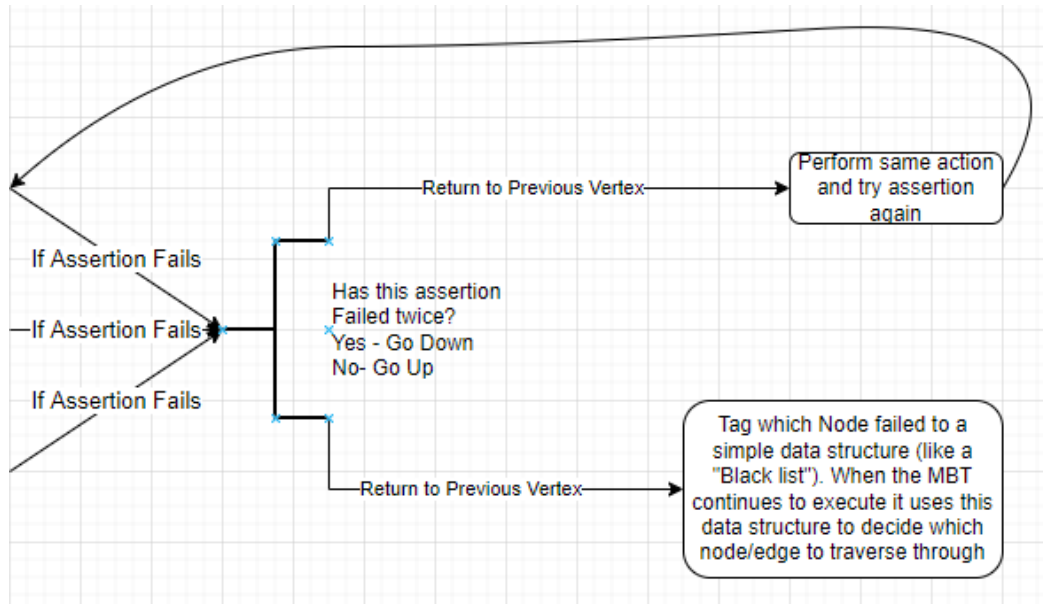


Figure 5 - GraphWalker With Additional Fault Tolerant Features

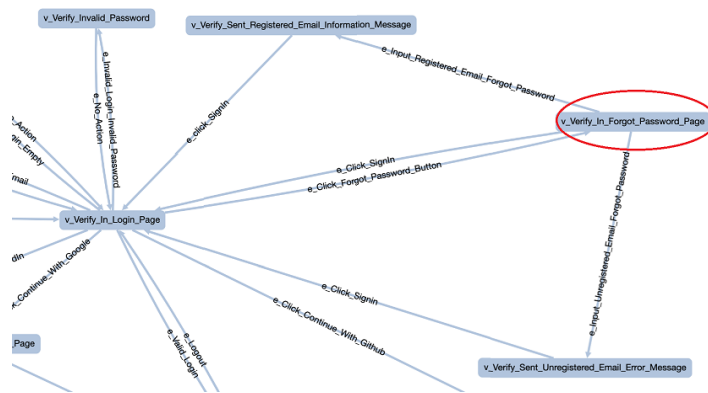
Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again

As seen in Figure 5. Graphwalker will first try to go back to the previous node and retry the action to get to the failed node. This is because some tests/assertions fail for seemingly no apparent reason, for example; failing due to the page loading its assets slowly, so your assertion couldn't find what it was looking for. These are called flaky tests, we know they are flaky if we repeatedly rerun these failed tests and they pass in some of the reruns.

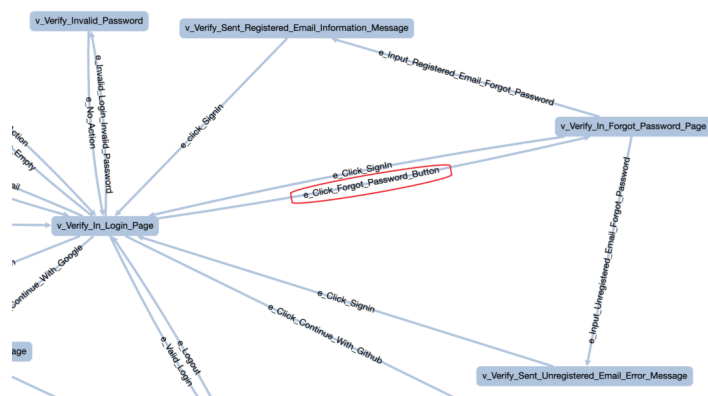
Instead of having to use the `ReplayMachine` class in the graphwalker codebase which reruns a previously executed path to try and see if the failure was reproducible, which takes at least double the time as you have effectively run the test twice. With this heuristic, it will reduce the amount of time taken as it only has to go back one node rather than repeat the whole executed path. This will hopefully minimise the negative effects of flaky tests which hinder development, slow down progress and can cost a lot of money to address.

As we don't want the node to be executed with no limits, we will have to design the system in such a way that the node that fails is marked, in such a way that the node has an upper limit of failing the assertion twice before the system stops using this heuristic. This could be as simple as a counter or some attribute that changes depending on the outcome of each assertion attempt.

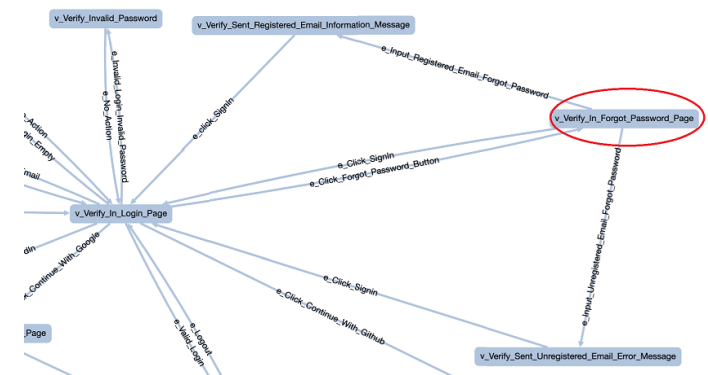
Example: Red circle being a failed node. Then the test tries to retake the edge leading up to the node followed by the reassertion of the failed node.



Step 1: Fails on Node



Step 2: Retakes edge leading to node



Step 3: Retry failed node

Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)

When a node fails, the test suite will have to give this node a label of sorts that lets the test suite know that this node has failed the assertion and that it should not try to traverse the edge to this node again.

The nodes and edges that are now impossible to access/unreachable due to not being able to traverse past this failed node, also need to be updated in a way that will allow the stop condition to be true eventually. For example, if the stop condition is 100% node coverage but you can't access a node due to not being able to traverse past the failed node, the test suite would never stop running. So some form of modification to either the stop condition and/or non accessible nodes and edges needs to be done.

The failed node needs to be assessed in terms of what nodes it blocks access to in the model the test suite is currently in and does it block the only shared state node, meaning the test suite can't access a single or multiple models because of that.

Once the nodes, edges and stop conditions have been updated if the stop condition has not already been met, the test suite needs to go back to the previous node and continue the test while avoiding the failed node.

Example below: Green circles are passed/covered nodes. Red circle is the node that has just failed. Orange circle would be the next element set. Purple circles are labelled as unreachable. Test would continue while avoiding the edge highlighted in red.



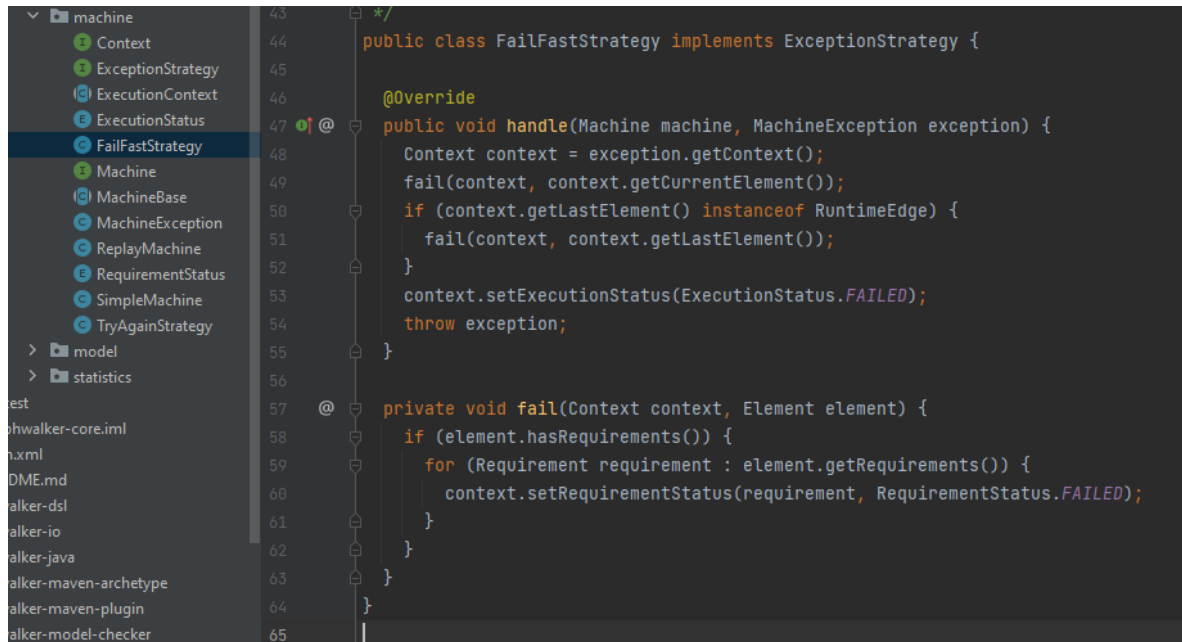
Node Fails



Unreachable nodes labelled. Orange circle taken back to covered node. Edge highlighted in red is avoided for the rest of the test execution

Implementation

Based on the design phase that was presented in the previous section, we can take a look at how to implement our proposed solution into the GraphWalker codebase [10]. Currently GraphWalker by default uses a class named `FailFastStrategy` as seen in figure 6 below.

The image is a screenshot of an IDE, likely IntelliJ IDEA, showing the source code of the `FailFastStrategy` class. On the left, a project explorer shows a package structure with folders like 'machine', 'model', and 'statistics', and various classes including `Context`, `ExceptionStrategy`, `ExecutionContext`, `ExecutionContext`, `FailFastStrategy` (which is selected), `Machine`, `MachineBase`, `MachineException`, `ReplayMachine`, `RequirementStatus`, `SimpleMachine`, and `TryAgainStrategy`. The main editor area shows the code for `FailFastStrategy` which implements `ExceptionStrategy`. The code includes an `@Override` annotation for the `handle` method, which takes a `Machine` and a `MachineException` as parameters. It retrieves the context, fails the current element, and if the last element is a `RuntimeEdge`, it fails that as well. It then sets the execution status to `FAILED` and throws the exception. A private `fail` method is also shown, which takes a `Context` and an `Element`, checks for requirements, and sets their status to `FAILED` if they exist.

```
43  */
44  public class FailFastStrategy implements ExceptionStrategy {
45
46      @Override
47      @Override
48      public void handle(Machine machine, MachineException exception) {
49          Context context = exception.getContext();
50          fail(context, context.getCurrentElement());
51          if (context.getLastElement() instanceof RuntimeEdge) {
52              fail(context, context.getLastElement());
53          }
54          context.setExecutionStatus(ExecutionStatus.FAILED);
55          throw exception;
56      }
57
58      @Override
59      private void fail(Context context, Element element) {
60          if (element.hasRequirements()) {
61              for (Requirement requirement : element.getRequirements()) {
62                  context.setRequirementStatus(requirement, RequirementStatus.FAILED);
63              }
64          }
65      }
66  }
```

Figure 6 - FailFastStrategy class

As you can see it implements the `ExceptionStrategy` class, which forces any class that implements it to make a handle method.

The idea is to create our own class which implements the `ExceptionStrategy` class as well, to implement our proposed design there. Before we get to the specifics of the class, there are several other concepts and changes that need to be made in order for our new class to be used instead of the current one.

One such change is in the `MachineBase` class where which strategy is going to be used is hard coded into the class. So this will be a simple switch of changing `FailFastStrategy` to our new class.

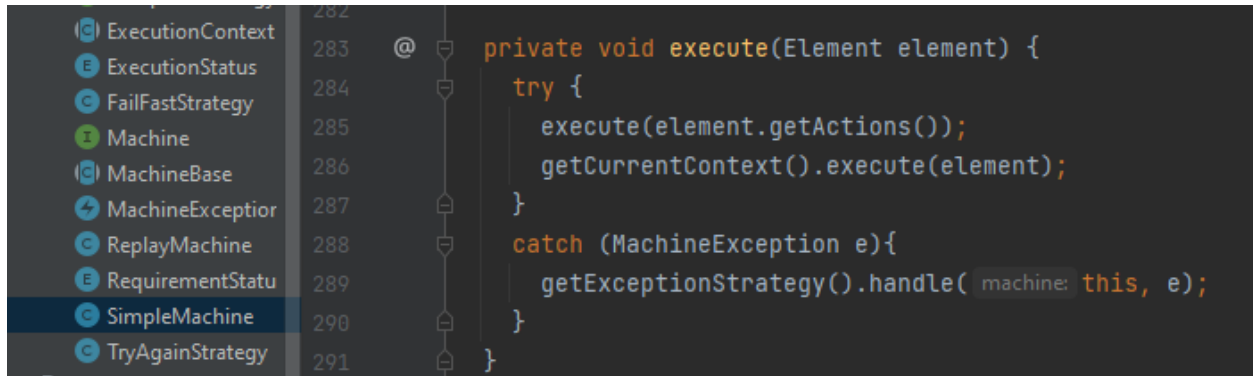


Figure 7 - execute method in SimpleMachine class

In figure 7 we see the execute method in the SimpleMachine class which is the working implementation of MachineBase. This is where the getExceptionStrategy() is called which is a method in the MachineBase class which will return the value in which the exceptionStrategy variable is set at which can be seen in the figure below.

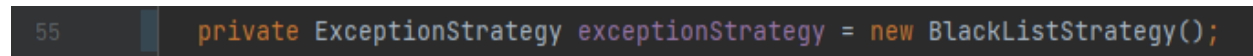


Figure 8 - Snippet of code about setting the exceptionStrategy variable from MachineBase class

A quick explanation on some information the system has when executing the SUT model. The system has a form of information called elements. An element can be either a node or an edge. There are three elements that the system has information on being; lastElement, currentElement and the nextElement. As we know, each model has a path generator, for example random which I talked about in the design section. These path generators decide by their internal logic which element should be the next currentElement to be executed if the nextElement is null. By changing the values of these elements, we can decide where the system perceives itself to be in the model and where it should traverse next.

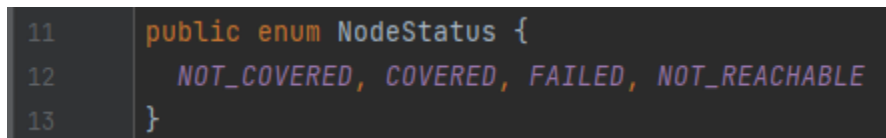


Figure 9 - NodeStatus class

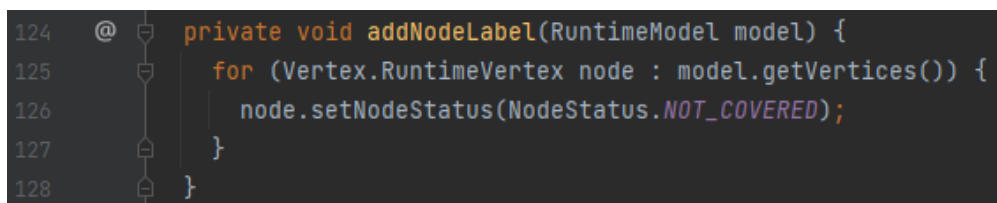


Figure 10 - Setting all nodes in a model to be uncovered at the start of execution

I have made my own enum `NodeStatus` as seen in figure 9 which is associated with every node in a model in order to keep track of its current situation within the test. Figure 10 shows a code snippet of a method that is called in the `ExecutionContext` class when a model is being instantiated. This will make sure that every node has the default value of `NOT_COVERED` when the test execution starts.

Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again

Firstly we will check whether this node has failed before or not. If it has, it means that this will be the second time this node has made the exception Strategy handle method be executed. When this happens the strategy will react in a similar way to the `FailFastStrategy` class which can be seen in Figure 6, which is to stop the test and bail out.

If this is the first time the node has made this class be called, then using the information we have available to us e.g what node failed and what the previous element is, in this case it's an edge. As we know the edge which is connected to the failed node, we can work backwards to find the previous node as each edge has a source and target node. We can set this source node to be the new current element and set the next element to be the edge connecting these two nodes. Doing this makes the system re-execute the edge that traverses towards the failed node, therefore giving it a second attempt at passing the assertion.

When setting the next element to be this connecting edge, the test may fail, as the edge will be trying to take an action that may not be possible from the current state of SUT. In this scenario we check that this edge is indeed trying to traverse to the failed node from before and therefore we allow it to continue without crashing the test and therefore allowing it to get to the failed node and retry the assertion. We do this as there may be times where the SUT may not have successfully left the previous node. Either way, it gives the SUT enough time to load the failed node's assets to be ready for the second assertion.

So unlike the `FailFastStrategy`, we won't be setting the `ExecutionStatus` to `FAILED`, as the `SimpleMachine` will not take another step if the `ExecutionStatus` is either `COMPLETED` or `FAILED`. When the `getExceptionStrategy` is called, the `ExecutionStatus` is already set to `FAILED`. So this will have to be set to `EXECUTING` before the system finishes executing the exception Strategy to keep the MBT test session running.

Finally, we label the failedNode with a Node status of FAILED, which is what enables us at the beginning of the class to check whether this is the node's first or second failure.

Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)

Firstly, the node which has failed is labelled as FAILED using the set node status method, as this will be useful later when the test is deciding what edges to take. This can be seen in the figure below.

```
28 if (context.getCurrentElement() instanceof Vertex.RuntimeVertex) {  
29     ((Vertex.RuntimeVertex) context.getCurrentElement()).setNodeStatus(NodeStatus.FAILED);  
}
```

Figure 11 - Setting the NodeStatus of the failed node to FAILED

As this heuristic involves deciding the reachability of nodes, we map all the nodes to their possible target and source nodes, which you can see the instantiation of in figure 12. The edges are then added one at a time to the appropriate key in the map.

```
32 HashMap<Vertex.RuntimeVertex, HashSet<Vertex.RuntimeVertex>> GTargetMap = new HashMap<>();  
33 HashMap<Vertex.RuntimeVertex, HashSet<Vertex.RuntimeVertex>> GSourceMap = new HashMap<>();
```

Figure 12 - Mapping Snippet from Heuristic 2

We do this in order to do a depth first search (DFS) through the makeshift graph. There are two other hashmaps that function as a lookup table which hold a boolean value based on whether we have either visited the node while searching and if we have determined if that node is reachable or not. True, being reachable and false being unreachable.

The hashmaps allow the program to only run through the makeshift graph fully once, as each call after the initial call is only accessing if the reachability hashmap has stored a True or False value. This is a form of top-down dynamic programming, as the heuristic is using both recursion and caching/memoization.

This heuristic is implemented this way due the graphs of each model being quite small so there isn't a likely possibility of a stack overflow error and also by storing the intermediate results of the DFS, we can minimise repetitive computations, therefore speeding up the test.

```

110 for(Vertex.RuntimeVertex source : GSourceMap.get(root)) {
111     isAnySourceReachable = source.getNodeStatus().equals(NodeStatus.COVERED) || isAnySourceReachable;
112     if (source.getNodeStatus().equals(NodeStatus.NOT_COVERED)) {
113         if (dfs(source, GTargetMap, GSourceMap, isVisited, mem)) {
114             isAnySourceReachable = true;
115         }
116     }
117 }

```

Figure 13 - Logic Snippet from Heuristic 2

The DFS method works initially by being provided a node which we call root in which the program recursively goes through the root's source node/s looking for a source node that has already been covered as seen in figure 13. Which lets us know the root and the nodes leading up to that root node are reachable and if we don't find a covered source node, we can say that the node is unreachable. We then go through all the target nodes of all the nodes we went through and do the same thing. Each time we find a covered source or the last source is uncovered, the hashmap lookup table is updated.

```

52 for (Vertex.RuntimeVertex vertex : allNodes) {
53     if (!dfs(vertex, GTargetMap, GSourceMap, isVisited, mem)) {
54         if (vertex.getNodeStatus().equals(NodeStatus.NOT_COVERED)) {
55             vertex.setNodeStatus(NodeStatus.NOT_REACHABLE);
56         }
57     }
58 }

```

Figure 14 - Logic to decide if the node status is updated from Heuristic 2

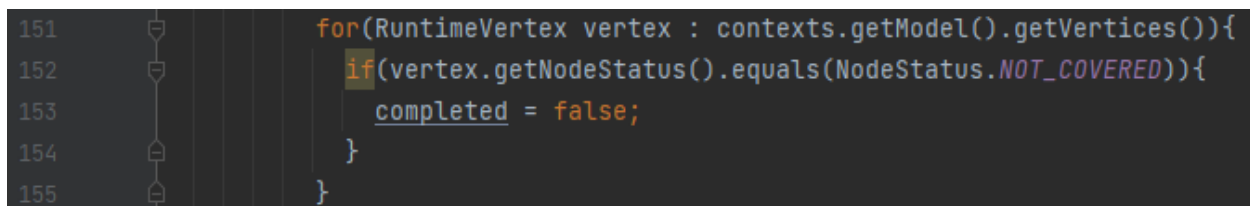
As seen in figure 14 when the DFS returns a false value we check to make sure that node is indeed not been covered and then change that node to a node status of NOT_REACHABLE.

Finally we set the execution status back to EXECUTING and then set the current element and next element to the appropriate elements depending on the situation.

This is one of 2 scenarios:

1. There is an edge that goes from the failed node to the previous node. So then we set the next element as that edge and then set the current element to the previous node as it won't change its state.
2. There is no edge that goes from the failed node to the previous node. So we set the current element to the previous node and let the system decide the next element outside of this heuristic.

To be more specific about how we can let the system decide the next elements for the rest of the test. Each time an element is decided to be the next element we make sure if it's an edge that it doesn't have a target node that is a failed node.



```
151 for(RuntimeVertex vertex : contexts.getModel().getVertices()){
152     if(vertex.getNodeStatus().equals(NodeStatus.NOT_COVERED)){
153         completed = false;
154     }
155 }
```

Figure 15 - Logic to decide if the test should stop executing from SimpleMachine class

As we cannot update the nodes and edges as visited due to not being able to access some of them, what we can do is assess if there are any nodes left not_covered as seen in figure 15, if there aren't any nodes that have not been covered for all models then we can say that the stop condition won't get any closer to being fulfilled and therefore we can stop the test after giving each model its appropriate execution status; which are either being completed if all the nodes are covered or failed if any of the nodes are failed or unreachable.

Summary of changed code in and added code to GraphWalker

Class	Code changed	Code added
NodeStatus.java		New enum class made solely by myself
TryAgainStrategy.java		New class made solely by myself implementing heuristic one.
BlackListStrategy.java		New class made solely by myself implementing heuristic two.
SimpleMachine.java	<ul style="list-style-type: none"> Line 139-194: Logic added to stop the test execution when certain criteria met. Logic added to set the execution status of models. Logic added to verify if the path generator's next step is allowed by the design of heuristic two, if not to try again. This does not impact heuristic one. Line 343-379: If these execute methods fail, they are now encapsulated in a try catch in order to allow the system to go to the heuristic that's been implemented. 	<ul style="list-style-type: none"> Line 60: global variable made called finalMove. Line 115: method call Line 129: method created
ExecutionContext.java		<ul style="list-style-type: none"> Line 107: method call Line 124-128: method setting all nodes in a model to NodeStatus.NOT_COVERED. Line 192-195: A method

		to set the next element, generally an edge, where the current element isn't set to null.
Vertex.java		<ul style="list-style-type: none"> Line 155-163: Added a nodeStatus to each node and made a getter and setter in order to change and review the status of each node.
Context.java		<ul style="list-style-type: none"> Line 101: Interface method.
MachineBase.java	<ul style="list-style-type: none"> Line 55: exceptionStrategy variable changed to point at the implemented heuristics. 	

Testing

As the design and implementation was developed for how the open-source MBT software GraphWalker reacts to faults, we will need to introduce faults into our SUT and then evaluate the fault tolerant heuristics based on how they handle these faults, i.e., whether the implementation of the two heuristics work as expected

Using the SUT Testinium, we can intentionally add a defect into one of the nodes. Using GraphWalker player, we will be able to see the test traverse into this node and then be able to evaluate how it performs afterwards.

```
public void v_Verify_Sent_Unregistered_Email_Error_Message() {  
    methods.checkElementVisible(methods.getBy( key: "unregisteredEmailMessageInForgotPassword"));  
    Assert.assertTrue( message: "", methods.doUrl( url: "https://account.testinium.com/uaa/send-reminder-mail",  
        count: 75, condition: "equal"));  
}
```

Figure 16 - Verification/Assertion on Node in login class

The defect could be made by changing the assert function in Figure 16 to something that will always fail, for example `assertTrue(1 == 2)`. This node is in the Login model as can be seen below. Prior to changing this assertion, we can see that the system can run through this model to 100% coverage rate with no failures.

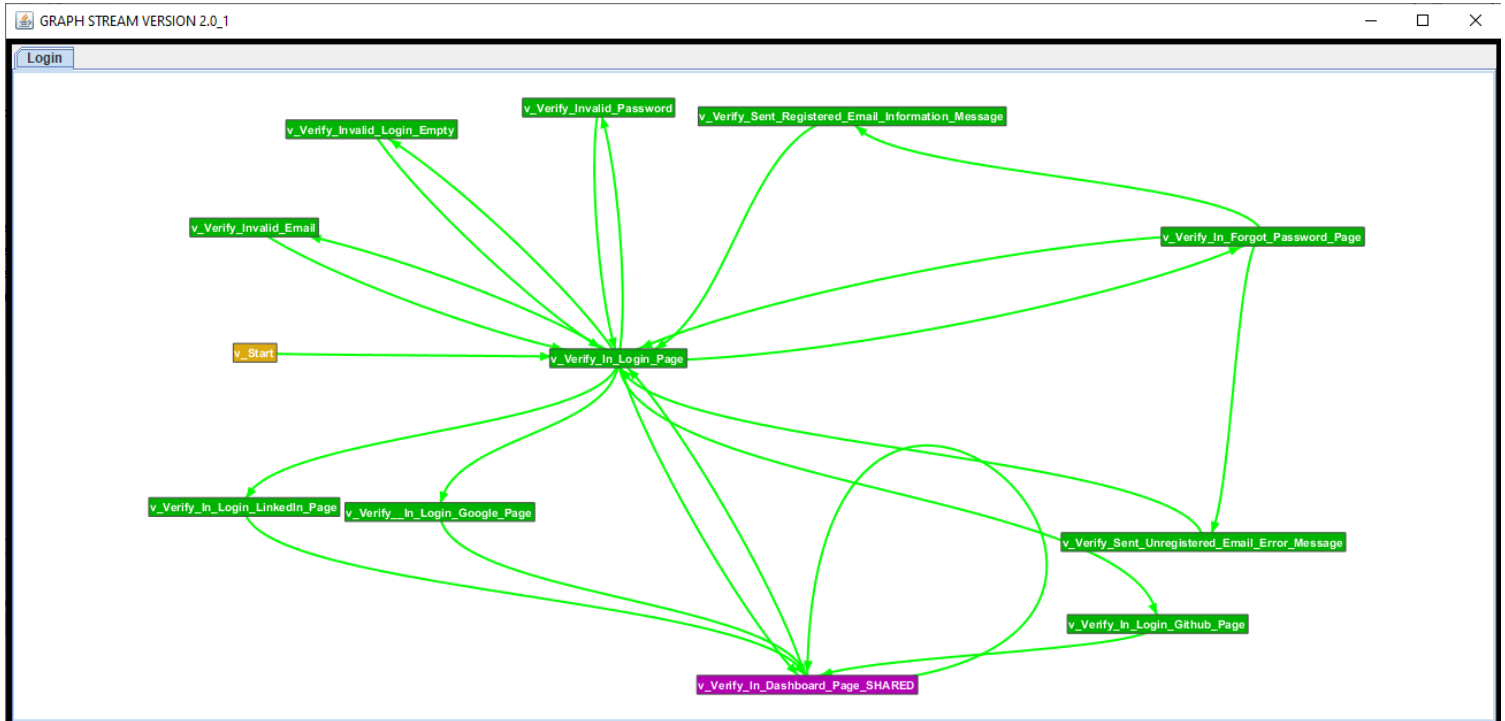


Figure 17 - Login Model for the SUT (Testinimum)

When we change this assertion to fail for example in the figure below and we run the same test as before.

```
public void v_Verify_Sent_Unregistered_Email_Error_Message() {
    methods.checkElementVisible(methods.getBy(key: "unregisteredEmailMessageInForgotPassword"));
    Assert.assertTrue(condition: 1 == 2);
    //Assert.assertTrue("", methods.doesUrl("https://account.testinium.com/uqa/send-reminder-mail",
    //    75, "equal"));
}
```

Figure 18 - Updated Assertion for node in Login class

We can see that when the test goes to that node it finds the condition to be false when it was expecting True. The test then stops executing immediately as the current GraphWalker core implementation is the original implementation which doesn't have the heuristics from the implementation phase added.

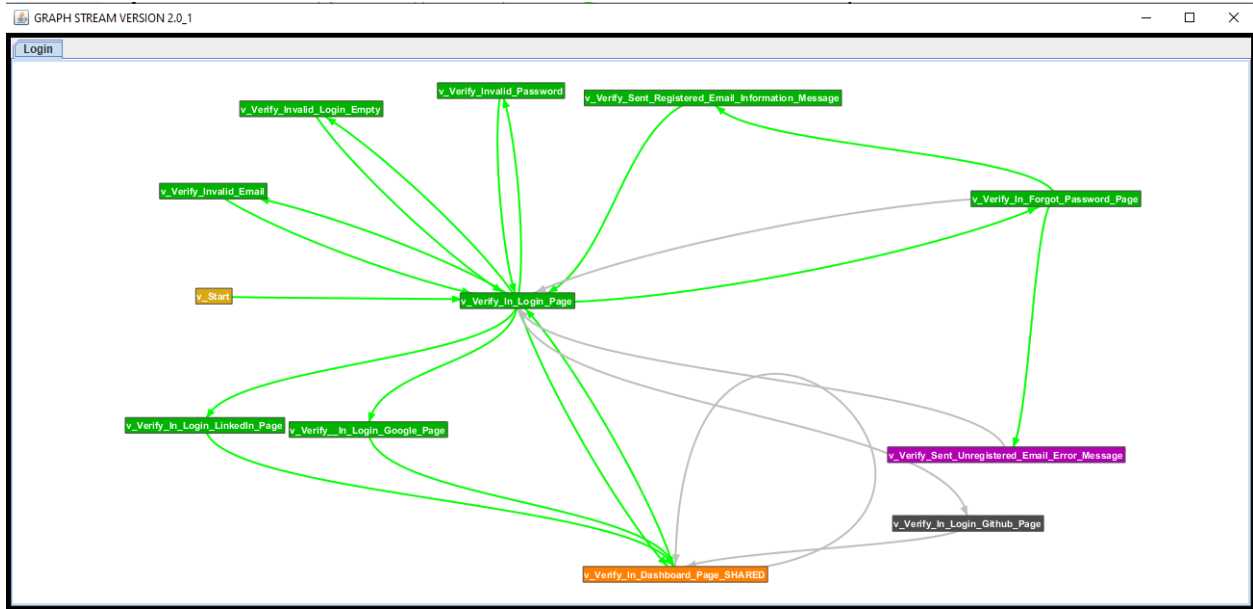


Figure 19 - Login Model for SUT (Testinium) with assertion errors on original graphwalker implementation

We can use the updated version of graphwalker by creating a .jar file of our graphwalker-core implementation and then setting that jar file in the pom.xml of our SUT test suite as the version we want to use. Then we can rerun the test and evaluate how it works with the conditions defined below.

Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again

Introducing flakey tests into our system isn't necessary in order to effectively test heuristic 1. To effectively test heuristic one, what is needed is a node to fail and then see how our implementation handles this failure.

For this heuristic, the success of the test will be evaluated on three conditions:

1. When the node fails, does it go back to the previous node?
(v_Verify_In_Forgot_Password_Page)
2. Does the test then execute the connecting edge followed up by the failed node?
(v_Verify_Sent_Unregistered_Email_Error_Message)
3. When the node fails for the second time, does the system halt the test?

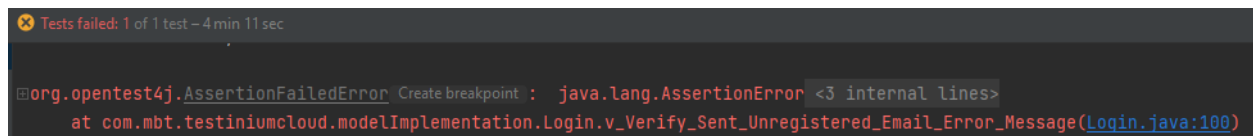
(yes it should)

To be sure that the heuristic is setting the path of this test correctly, by going to the correct edge and therefore node from the previous node. The test will be done multiple times to ensure that by chance the model hasn't randomly selected an edge that connects to the previous node which happens to be the edge we are evaluating success upon. In order to say this heuristic is successful it must do all 3 conditions correctly every time.

The expectation of this heuristic is to find the assertion at `v_Verify_Sent_Unregistered_Email_Error_Message` to be false but rather than stopping the execution of the SUT it will go back to the node `v_Verify_In_Forgot_Password_Page` and then traverse back over the edge to retry `v_Verify_Sent_Unregistered_Email_Error_Message`. If it then finds the assertion to be false again which it should, it will then like the previous implementation of GraphWalker, stop the execution of the SUT and deliver the message that the test failed at this particular node.

Test number:	Did the node go back to the previous node	Does the test suite then execute the edge which is connected to the failed node	Does the system then halt the test
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	Yes	Yes	Yes

Figure 20 - Heuristic 1 test results

A screenshot of a terminal window showing a test failure. At the top, a status bar says "Tests failed: 1 of 1 test - 4 min 11 sec". Below it, the error message is displayed in red and white text: "org.opentest4j.AssertionFailedError Create breakpoint : java.lang.AssertionError <3 internal lines> at com.mbt.testiniumcloud.modelImplementation.Login.v_Verify_Sent_Unregistered_Email_Error_Message(Login.java:100)".

```
Tests failed: 1 of 1 test - 4 min 11 sec
org.opentest4j.AssertionFailedError Create breakpoint : java.lang.AssertionError <3 internal lines>
at com.mbt.testiniumcloud.modelImplementation.Login.v_Verify_Sent_Unregistered_Email_Error_Message(Login.java:100)
```

Figure 21 - Test Heuristic 1 fails

Figure 21 and 22 shows the defect that was put in has successfully failed the assertion resulting in the test failed at that node. The exact line (100) of failure being shown in figure 27.

Step 1:

At the start of exception Class

```
> f executionStatus = {ExecutionStatus@11021} "FAILED"
v f currentElement = {Vertex$RuntimeVertex@11022}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11029} "NOT_COVERED"
  > f id = "00b41950-48a8-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Unregistered_Email_Error_Message"
```

Figure 22 - Failed node in login model for Heuristic 1

Step 2:

At the end of exception
Class

```
> f executionStatus = {ExecutionStatus@11048} "EXECUTING"
v f currentElement = {Vertex$RuntimeVertex@11045}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11051} "COVERED"
  > f id = "fd9ca020-48a7-11ea-8909-ddb066462915"
  > f name = "v_Verify_In_Forgot_Password_Page"
  f actions = {Collections$UnmodifiableRandomAccessList@11054} size = 0
  f requirements = {Collections$UnmodifiableSet@11055} size = 1
  f properties = {Collections$UnmodifiableMap@11056} size = 2
v f nextElement = {Edge$RuntimeEdge@11023}
  > f sourceVertex = {Vertex$RuntimeVertex@11045}
  > f targetVertex = {Vertex$RuntimeVertex@11022}
  > f guard = {Guard@11062}
  > f weight = {Double@11063} 0.0
  > f dependency = {Integer@11064} 0
  > f id = "70b8d3f0-48ab-11ea-8909-ddb066462915"
  > f name = "e_Input_Unregistered_Email_Forgot_Password"
  f actions = {Collections$UnmodifiableRandomAccessList@11067} size = 0
  f requirements = {Collections$UnmodifiableSet@11068} size = 0
  f properties = {Collections$UnmodifiableMap@11069} size = 0
v f lastElement = {Vertex$RuntimeVertex@11022}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11040} "FAILED"
  > f id = "00b41950-48a8-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Unregistered_Email_Error_Message"
```

Figure 23 - Context of the SUT suite, specifically Execution Status, current element, next element and last element
for Heuristic 1 test

Step 3:

Executing edge that connects the passed node and the failed node

```
> f executionStatus = {ExecutionStatus@11048} "EXECUTING"
v f currentElement = {Edge$RuntimeEdge@11023}
  > f sourceVertex = {Vertex$RuntimeVertex@11045}
  > f targetVertex = {Vertex$RuntimeVertex@11022}
  > f guard = {Guard@11062}
  > f weight = {Double@11063} 0.0
  > f dependency = {Integer@11064} 0
  > f id = "70b8d3f0-48ab-11ea-8909-ddb066462915"
  > f name = "e_Input_Unregistered_Email_Forgot_Password"
```

Figure 24 - Connecting edge being the current element for Heuristic 1 test

Step 4:

Re-execute failed node

```
> f executionStatus = {ExecutionStatus@11048} "EXECUTING"
v f currentElement = {Vertex$RuntimeVertex@11022}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11040} "FAILED"
  > f id = "00b41950-48a8-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Unregistered_Email_Error_Message"
```

Figure 25 - Failed node being executed for a 2nd time for Heuristic 1 test

Step 5:

End execution of SUT

```
> f executionStatus = {ExecutionStatus@11021} "FAILED"
v f currentElement = {Vertex$RuntimeVertex@11022}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11040} "FAILED"
  > f id = "00b41950-48a8-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Unregistered_Email_Error_Message"
```

Figure 26 - Failed node failed for a 2nd time for Heuristic 1 test

```
97 | public void v_Verify_Sent_Unregistered_Email_Error_Message() {
98 |
99 |     methods.checkElementVisible(methods.getBy( key: "unregisteredEmailMessageInForgotPassword"));
100 |     Assert.assertTrue( condition: 1 == 2);
```

Figure 27 - Login class V_Verify_Sent_Unregistered_Email_Error_Message assertion for Heuristic 1 test

Figure 23 shows that the current element was successfully changed to the previous node as well as the next element to be executed to be the adjoining edge of the previous and failed node. You

can see the source and target vertex of the edge (figure 24) has the same values as the current node and failed node in figure 23.

As shown in figure 25, the failed node is then re-executed and then proceeds to fail again which can be seen in figure 26. This ends the execution which is shown in figure 21.

Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)

For testing this heuristic there were several situations that could be looked into:

1. A node fails and the outgoing edges from that node are the only edges that go into the nodes that the edges are traversing to.
2. A node fails and the outgoing edges from that node are not the only edges that go into that node that the edges are traversing to:
 - a. All the edges originate from failed nodes
 - b. A mixture of failed and not covered nodes
 - c. At least 1 edge originates from a passed node

When looking at the implementation of the heuristic, it doesn't differentiate between the several situations. So we can test situation one and then extrapolate on that to presume the other situations will handle in a similar fashion. Using the previous login model from the SUT (figure 17) we can test situation one. The node `v_Verify_In_Forgot_Password_Page` will have a defect in the assertion so that it fails.

We will evaluate the implementation of the heuristic based on these four conditions:

1. When the node fails does it go back to the previous node?
We expect the test suite to take the edge from `v_Verify_In_Forgot_Password_Page` to `v_Verify_In_Login_Page`.
2. Does the test ever try to execute an edge or node that leads to or is already classified as FAILED?
We expect this to not happen.
3. Does the test stop executing?
We expect the test to stop once all accessible nodes have been traversed.
4. Are the correct nodes labelled as failed and Not_Reachable?
Nodes that are expected to be labelled as failed are as follows:

v_Verify_In_Forgot_Password_Page.

Nodes that are expected to be labelled as unreachable are as follows:

v_Verify_Sent_Unregistered_Email_Error_Message

and

v_Verify_Sent_Registered_Email_Information_Message

Test number:	Did the node go back to the previous node?	Did the test try to execute a failed node or edge that leads to a failed node?	Did the test stop executing?	Are the correct nodes labelled as failed and Not_Reachable?
1	Yes	No	Yes	Yes

Figure 28 - Heuristic 2 test results

Step 1: Node fails and calls the Exception Class

```
> f executionStatus = {ExecutionStatus@11017} "FAILED"
v f currentElement = {Vertex$RuntimeVertex@11018}
  f sharedState = null
> f nodeStatus = {NodeStatus@11025} "NOT_COVERED"
> f id = "fd9ca020-48a7-11ea-8909-ddb066462915"
> f name = "v_Verify_In_Forgot_Password_Page"
```

Figure 29 - Failed node in login model for Heuristic 2

Step 2: End of exception class

```
allNodes = {Collections$UnmodifiableRandomAccessList@11006} size = 12
> 0 = {Vertex$RuntimeVertex@11094}
> 1 = {Vertex$RuntimeVertex@11041}
> 2 = {Vertex$RuntimeVertex@11095}
> 3 = {Vertex$RuntimeVertex@11018}
v 4 = {Vertex$RuntimeVertex@11096}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11104} "NOT_REACHABLE"
  > f id = "fe596070-48a7-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Registered_Email_Information_Message"
  f actions = {Collections$UnmodifiableRandomAccessList@11107} size = 0
  > f requirements = {Collections$UnmodifiableSet@11108} size = 1
  > f properties = {Collections$UnmodifiableMap@11109} size = 2
v 5 = {Vertex$RuntimeVertex@11097}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11104} "NOT_REACHABLE"
  > f id = "00b41950-48a8-11ea-8909-ddb066462915"
  > f name = "v_Verify_Sent_Unregistered_Email_Error_Message"

> f executionStatus = {ExecutionStatus@11042} "EXECUTING"
v f currentElement = {Vertex$RuntimeVertex@11041}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11053} "COVERED"
  > f id = "f9c384a0-48a7-11ea-8909-ddb066462915"
  > f name = "v_Verify_In_Login_Page"
  f actions = {Collections$UnmodifiableRandomAccessList@11056} size = 0
  > f requirements = {Collections$UnmodifiableSet@11057} size = 1
  > f properties = {Collections$UnmodifiableMap@11058} size = 2
v f nextElement = {Edge$RuntimeEdge@11081}
  > f sourceVertex = {Vertex$RuntimeVertex@11018}
  > f targetVertex = {Vertex$RuntimeVertex@11041}
  > f guard = {Guard@11086}
  > f weight = {Double@11087} 0.0
  > f dependency = {Integer@11047} 0
  > f id = "22ce14c0-48ab-11ea-8909-ddb066462915"
  > f name = "e_Click_SignIn"
  f actions = {Collections$UnmodifiableRandomAccessList@11090} size = 1
  f requirements = {Collections$UnmodifiableSet@11091} size = 0
  f properties = {Collections$UnmodifiableMap@11092} size = 0
v f lastElement = {Vertex$RuntimeVertex@11018}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11038} "FAILED"
  > f id = "fd9ca020-48a7-11ea-8909-ddb066462915"
  > f name = "v_Verify_In_Forgot_Password_Page"
```


Figure 30 - Execution status updated, unreachable nodes updated, failed node updated, current and next element set

Step 3: Execute the edge leading back to the covered node

```
> f executionStatus = {ExecutionStatus@11042} "EXECUTING"
v f currentElement = {Edge$RuntimeEdge@11081}
  > f sourceVertex = {Vertex$RuntimeVertex@11018}
  > f targetVertex = {Vertex$RuntimeVertex@11041}
  > f guard = {Guard@11086}
  > f weight = {Double@11087} 0.0
  > f dependency = {Integer@11047} 0
  > f id = "22ce14c0-48ab-11ea-8909-ddb066462915"
  > f name = "e_Click_SignIn"
```

Figure 31 - Connecting edge leading to previous node

Step 4: Successfully reach covered node

```
> f executionStatus = {ExecutionStatus@11042} "EXECUTING"
v f currentElement = {Vertex$RuntimeVertex@11041}
  f sharedState = null
  > f nodeStatus = {NodeStatus@11053} "COVERED"
  > f id = "f9c384a0-48a7-11ea-8909-ddb066462915"
  > f name = "v_Verify_In_Login_Page"
```

Figure 32 - Covered Node execution

Step 5: Test suite continues execution on other edges that don't lead to failed nodes

```
> f executionStatus = {ExecutionStatus@11042} "EXECUTING"
v f currentElement = {Edge$RuntimeEdge@11176}
  > f sourceVertex = {Vertex$RuntimeVertex@11041}
  v f targetVertex = {Vertex$RuntimeVertex@11099}
    f sharedState = null
    > f nodeStatus = {NodeStatus@11025} "NOT_COVERED"
    > f id = "278221c0-48bd-11ea-8909-ddb066462915"
    > f name = "v_Verify_Invalid_Email"
```

Figure 33 - Edge execution after failure of a node in the model

Step 6: Test result - 1 Failed model, unreachable nodes labelled as not visited

```
✓ Tests passed: 1 of 1 test - 21 min 51 sec
{
  "totalFailedNumberOfModels": 1,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 2,
  "verticesNotVisited": [
    {
      "modelName": "Login",
      "vertexName": "v_Verify_Sent_Registered_Email_Information_Message",
      "vertexId": "fe596070-48a7-11ea-8909-ddb066462915"
    },
    {
      "modelName": "Login",
      "vertexName": "v_Verify_Sent_Unregistered_Email_Error_Message",
      "vertexId": "00b41950-48a8-11ea-8909-ddb066462915"
    }
  ]
}
```

Figure 34 - Test Heuristic 2 results

As we can see in figure 29, the expected node does indeed produce an assertion failure and therefore our heuristic is called. We then can see from figure 30 that the failed node is labelled as failed as well as the now unreachable nodes labelled as NOT_REACHABLE. Then the current and next element are set correctly so that we can traverse back to the passed/covered node.

In figure 32 and 33, we can see that traversal in more detail. With the test continuing execution on with the rest of the model while avoiding the failed node. Once all reachable nodes have been covered the test ends, the results are then shown which can be seen in figure 34. The important properties to note, is that the number of failed models is correct and the vertices not visited match the nodes which were labelled as unreachable.

System Evaluation and Experimental Results

Overall this project was a success, in that, if in the future I need to use GraphWalker I would definitely use my fault tolerant heuristics over what's currently available and I believe others would also use my implementation of these fault tolerant features.

These heuristics do meet the mark in many places and when looking at the tests that were made in order to evaluate if these goals were met, they certainly do in many places. The code is concise, legible and reliable but when it comes to code modifiability and accessibility, it is lacking due to time constraints of the project. Further testing of the heuristics on the SUT as well as other SUT's would have raised my confidence to a higher level.

One good point to note, is that the implementation of the heuristics don't affect each other in other classes that they both have to use. For example, one property for heuristic 2 is that the path generator cant use an edge that points to a failed node, which is exactly what heuristic 1 does. But the `simpleMachine` class code has been made in a way that only when `next_element` is null, that this condition is evaluated on it. Therefore no problems happen when changing from one heuristic to another due to `next_element` being set in heuristic 1 in the exception class. In the appendix section there is a guide on how to change the code to run either heuristic.

Fault-tolerance heuristic 1: Go back to the previous node and re-execute the failed node / edge again

For this heuristic, the project was an overall success. The goal was to give a node which had failed a second chance of passing the assertion before defaulting back to the original exception strategy of halting the execution of the test. This heuristic does this by immediately taking the edge that connects the previous node and the failed node, therefore producing an action which connects this covered node to the failed node. Then if the same node fails again it uses the same method of halting the execution of the test that the original implementation used. This can all be seen in figure 21-27 in the previous section (testing).

Even with the lack of flexibility in deciding the amount of chances a node should get before the test suite decides to stop executing, the current implementation is still more worthwhile than using the default strategy implemented in Graphwalker. As giving the nodes a second chance in order to possibly pass a flakey test is better than only having one chance. Due to the heuristic immediately re-executing the edge and then failed node there is no downside that comes with it,

for example time spent; as each individual edge/node does not take a large amount of time to re-execute especially when compared to the amount of time a flakey test could end up wasting. Without this heuristic, to try to reproduce a failure to determine if the failure was flaky or not would require a re-run of a previous execution from start to finish, which would clearly take much longer than re-executing two elements.

Fault-tolerance heuristic 2: Go back to the previous node and continue the MBT execution to other nodes (Black listing the failed nodes/edges)

This heuristic is a lot more complicated than the previous one, as it involves graph theory problems, verification of the elements in which the path generators produce (which decides which node/edge will be traversed over next) and how the stop condition is evaluated. The results produced and shown in figures 28-34 are something I am pleased with. In that the model not only successfully goes back to the previous node, but also continues testing until the stop condition cannot get any more fulfilled, while avoiding execution of the edge that leads to the failed node. It is also important that the implementation is implemented in such a way that it not only works but that it is doing so in a time efficient manner.

The top-down dynamic programming approach that I took was used due to this problem really being an optimisation problem involving graph theory. I feel that the mapping of the nodes to its target and source nodes should really be a whole OO class and data structure of its own. The ability to know how nodes and edges are linked to each other over a whole model or models is quite lacking in GraphWalker. Currently, this was made inside the exception class but that means it has to be made from scratch each time the exception class is called which isn't an efficient use of resources.

A note on the top condition is that it evaluates all the models in the test and only when there are no more uncovered nodes does it stop. Once it has decided to stop it correctly labels models with either passed or fails depending on whether there are any failed nodes within them.

Possible Future Improvements

At the start of this module, I was undertaking a different project under a different supervisor but due to a number of reasons that were out of my control, I had to switch, both my project and supervisor. In the middle of February I took on this project which inevitably gave me time constraints. Due to this, the ability to pick which heuristic you (as a user) would like to use for your tests is not as time efficient and effortless as one would like it to be. Current implementation means you would have to access the source code and change the exception strategy variable in the `MachineBase` class to point at the class you want it to use. Then you would package that source code into a jar file and then replace the jar file that you were using previously and then you can run your tests.

Places where heuristic 1 could have been improved on, is that the code is designed in such a way that it does not allow for the user to personally decide how many times they wish to give a node a chance at succeeding at each assertion. For example if the user decided they wanted each node to have four chances instead of the two currently given they would not be able to quickly implement this in a clean way. This is because the nodes that fail are given a label that allows the system to recognise this label if it fails again, which prompts the termination of the test. Instead of some form of numeric counter that the system could update every time it fails and only stop once a user defined limit has been reached.

Heuristic 2 could have more work done surrounding what happens when a node that is a shared state fails. As the nodes that now may be unreachable could be in a different model. The thought process that needs to be taken around this problem would be:

- Is that the only shared state in the model?
 - a. If yes, then you can simply set the model in which that node is shared to as unreachable.
 - b. If not, Another question arises, can you get to the model that the failed node was shared to through a different way (which would involve traversing through several models which involves the other shared nodes)
 - i. If so, then that model is reachable

- ii. If not, then yes that model would be unreachable

I would also look into making the mapping of the individual models into a larger map of connected models outside the fault tolerant classes as a permanent feature. And finally, add a visual colour for failed and unreachable nodes.

References

- [1]Glen O'Donovan. (2022). CSC3002 Project Available:
<https://gitlab2.eecs.qub.ac.uk/40233306/glenfinalyearproject>
- [2] V. Garousi and F. Elberzhager, "Test automation: not just for test execution," IEEE Software, vol. 34, no. 2, pp. 90-96, 2017.
- [3] M. Utting and B. Legeard, Practical model-based testing: a tools approach. Elsevier, 2010.
26
- [4] W. Elmendorf, "Automated design of program test libraries," IBM Technical report, TR 00.2089,
<https://benderrbt.com/Automated%20Design%20of%20Program%20Test%20Libraries%20-%201970.pdf>, 1970.
- [5]Kristian Karl. (2019). GraphWalker. Available: <https://graphwalker.github.io>.
- [6]testinium.io
- [7]Selenium. (2011). Selenium Java Maven Plugin. Available:
<https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java>.
- [8]Kristian Karl. (2020). Test path generation:
<https://github.com/GraphWalker/graphwalker-project/wiki/Test-path-generation>
- [9]Vahid Garousi, Alper Buğra Keleş, Yunus Balaman, Zeynep Özdemir Güler, and Andrea Arcuri, "Model-based testing in practice: An experience report from the web applications domain," Journal of Systems and Software, vol. 180, paper ID: 111032, October 2021
- [10] KristianKarl. (2021) Graphwalker Core
<https://github.com/GraphWalker/graphwalker-project/tree/master/graphwalker-core>

[11] Dr. Vahid Garousi, Bahar Software Engineering Consulting Corporation, UK; and Queen's University Belfast, UK Alper Buğra Keleş, and Yunus Balaman, Saha Information Technologies A.Ş., Turkey

<https://github.com/vgarousi/MBTofTestinium>

[12]Spring. (2020). PetClinic. Available: <https://github.com/spring-projects/spring-petclinic> . Last accessed 19th April 2022.

[13]Kristian Karl. GraphwalkerGithub.
<https://github.com/GraphWalker/graphwalker-project/releases>

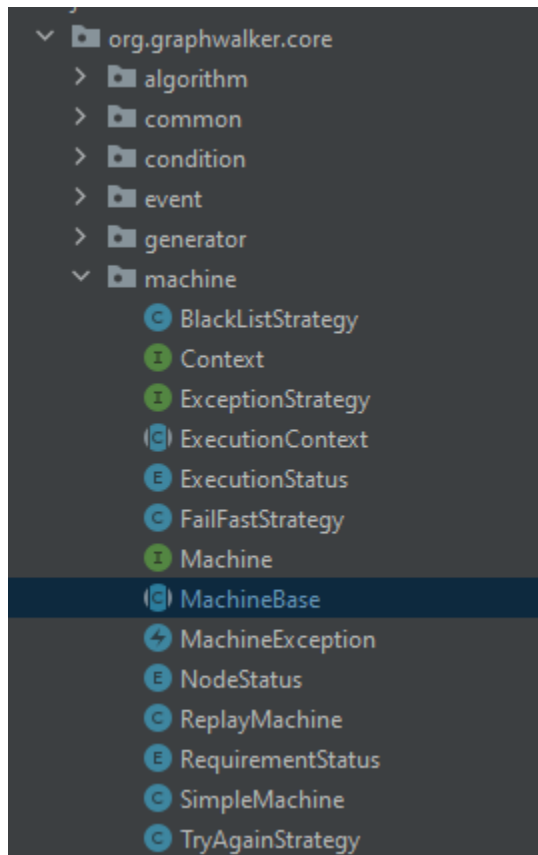
[14] M. Polo, P. Reales, M. Piattini, and C. Ebert, "Test automation," IEEE software, vol. 30, no. 1, pp. 84-89, 2013.

Appendices

User guide on how to switch from one heuristic to another:

As this project is going to be released as open-source on my github account, as well as linked on my project supervisor's github, I will make a guide to show how to switch between heuristics, to make the switch easier than looking through the source code.

1:Download my implementation of the source code for graphwalker-core.

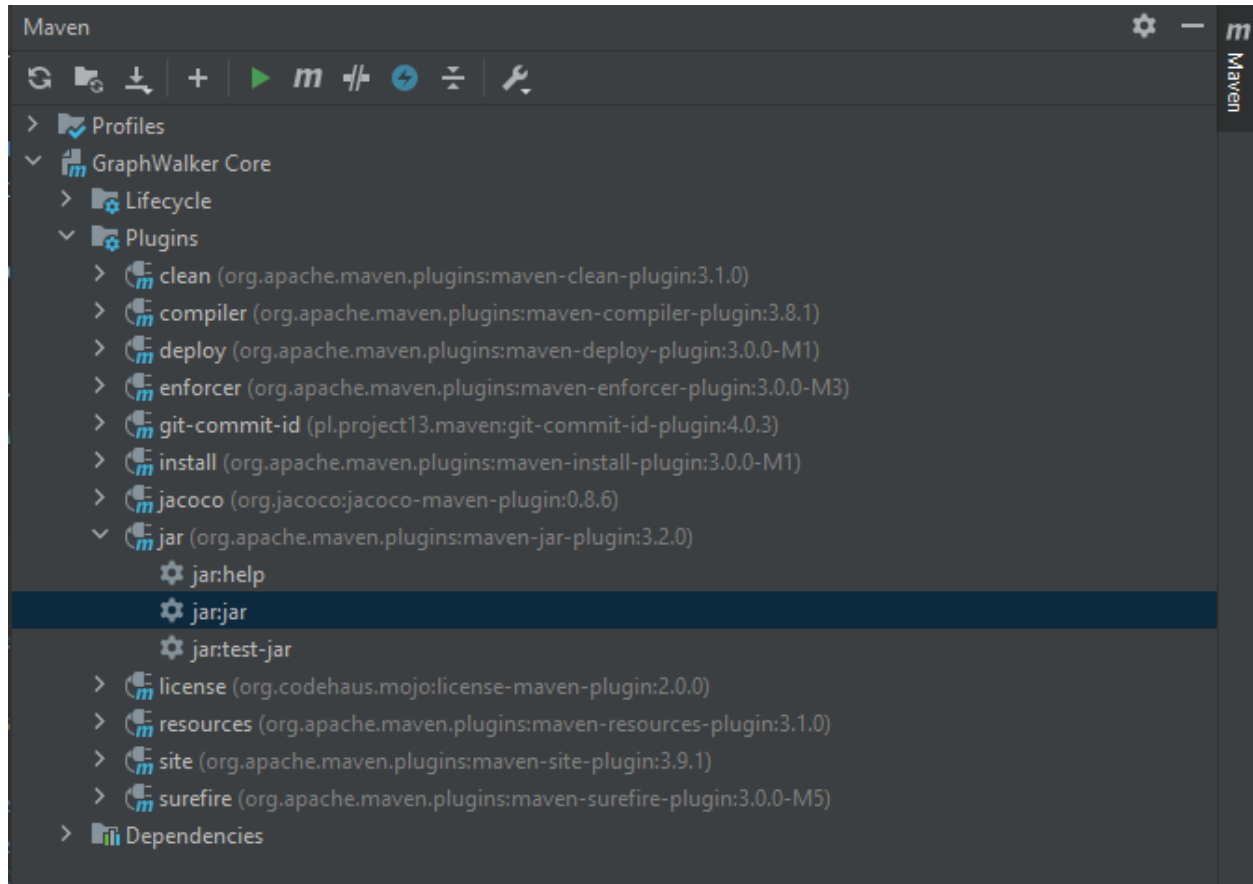


2:Go to the MachineBase class.

```
49  public abstract class MachineBase implements Machine {  
50  
51      private final List<Context> contexts = new ArrayList<>();  
52      private final List<Observer> observers = new ArrayList<>();  
53      private final Profiler profiler = new SimpleProfiler();  
54  
55      private ExceptionStrategy exceptionStrategy = new BlackListStrategy();
```

3: In this class you will see the variable `exceptionStrategy`. Change this variable to either what it's currently showing in the picture above or `TryAgainStrategy` if you want heuristic 1 or if you want to use the default strategy you can set it to `FailFastStrategy`.

4: I find it best to recompile the class and then reload all maven projects.



5: Here I am using IntelliJ, click on the Maven widget found in the top right corner, open the jar section and double click `jar:jar`

```
<dependency>
  <groupId>org.graphwalker</groupId>
  <artifactId>graphwalker-core</artifactId>
  <version>4.3.1</version>
  <scope>system</scope>
  <systemPath>${project.basedir}/graphwalker-core-4.3.1.jar</systemPath>
</dependency>
```

6: A jar file will be created, move this to where your pom.xml in your test suite is pointing at graphwalker-core. Above is how my pom.xml file looks like. So my graphwalker-core jar file would be in my test suite home directory. Then you are ready to run your tests.