MBTModelGenerator:

A software tool for automated generation of Model-based Test (MBT) models for web applications using clickstream data

Technical report, including system requirements, design, implementation, testing and evaluation of the tool

Tool website: github.com/vgarousi/MBTModelGenerator

Sasidhar Matta
Queen's University Belfast, Belfast, UK
smatta01@qub.ac.uk

Vahid Garousi
Bahar Software Engineering Consulting Limited, UK
Queen's University Belfast, Belfast, UK
v.garousi@qub.ac.uk

Last updated: September 28, 2022

Abstract:

Automated testing has now become mainstream in the software industry. However, development of automated test suites is still effort-intensive. Thus, many software engineering practitioners and researchers have been exploring different ways to design/ derive automated test cases in fully or partially automated manners.

Clickstream data are one source of such approaches. A clickstream or click-path is the sequence of hyperlinks one or more website visitors follows on a given website, presented in the order viewed. Clickstream data can be mined and visualized as clickstream graphs. Tools such as Google Analytics also provide simple visualization of clickstream data, e.g., it is called "Behavior Flow" in Google Analytics. Furthermore, various success stories of using clickstream data for test generation by various software companies has been reported.

In this project, we have designed and developed a software tool which would receive the clickstream data of a given web application under test and will generate, as output, the corresponding clickstream graph. The output graph should be in the form of a state-chart in the widely-used Model-based Testing (MBT) approach. In terms of file format, the state-chart (test model) shall be generated in the file format of a widely-used MBT tool called GraphWalker (graphwalker.github.io), e.g., we should be able to open the model in GraphWalker directly. It is expected that the generated test model may not be perfect or as precise as a human software engineer would design, due to having impartial information in clickstream data, and thus, it is expected that the user (a test engineer in this case) would need to "fix" or make improvements to it, to make the model more meaningful or then to later execute / run the test model in GraphWalker..

The tool that is being demonstrated in this project records this click stream data and creates a MBT graph file which is later used by the automated testing tool GraphWalker to automatically test the SUT.

TABLE OF CONTENTS

1 INTRODUCTION

Testing of software is as important as development of the software and in order to speed up the development, there exists several software development methodologies and testing methodologies too, but development cannot be automated whereas testing can be automated.

The automated tests perform a series of actions or calls on the software under test (SUT) and verify the results against a pre-determined set of correct values to verify if the software is working correctly.

However, the automated tests are only limited to tasks such as Unit tests, Integration tests where all the tests to be performed are pre-determined and manually coded by a tester. This has to be done manually as the code's backend may not have a physically interactable user interface.

In case of a website, which is also a software, the main part is a user interface which can be physically interactable by a human and can be tested manually. But it can be quite a hassle as a human is needed to test it which has its advantages and disadvantages where the advantages are: Depth of testing – A human can go into depths of the website and can also randomly test different components of a website. Tests done will also be close to that of normal usage by the clients which are true to natural usage patterns. The disadvantages are: Lack of speed – A machine can perform tests at exceeding speeds compared to a human.

The problem that we will be working on this project is related to automated testing of websites based on the click-stream data received from a website (SUT).

Click-stream data is a recording of all interactions such as clicks and textual interactions between the user and the software while it is being used. This click-stream data is mostly used in analysis. But in this project, we aim to use this stream data to generate a automated test in the format of a Model Based Test (MBT) which can later be replayed as an automated test using the GraphWalker library which will automatically navigate through the user interface based on the MBT file generated and validate the results against a predetermined set of outcomes.

2 SYSTEM REQUIREMENTS AND SPECIFICATION

SYSTEM REQUIREMENTS

The System requirements were to design a system that is capable of receiving clicks and page load data and convert them into an automatically testable data format. This is similar to that of a recording session where all the actions done by the tester are recorded once and are then replayed step by step.

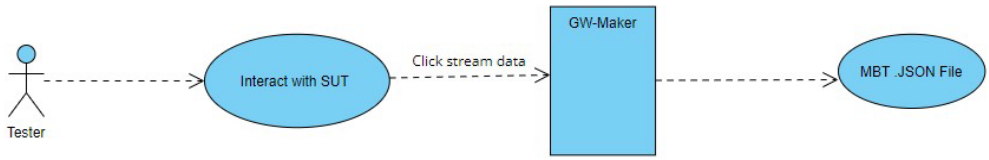

Figure 1: User case diagram of Graph Walker Maker Tool.

But a challenge is faced when trying to record this session as all the inputs have to perfectly recorded such as the mouse clicks and individual input events. But, since all the requests from the client are sent to the server are sent in HTTP request format. These can be captured instead or even better; we can capture the website events on the client computer instead and record these events in a serial manner. This is the approach we are taking to obtain the input events.

The click steam data is the series of events such as the click events on the various UI elements on the client side and the page loads are concentrated into a single data store. This data can later be utilized for various purpose such as user testing, security audits and in the latest times, it can also be used to automate the testing. When collected, this click stream data can be replayed using certain tools to automate the testing.

The current aim of this project is to collect the click stream data into a Model Based Test (MBT) formatted JSON file which can then be used to automatically test the same website SUT (System Under Test) by reading this JSON file. The MBT format consists of two basic data types which are Edges and Vertices. An example MBT format that is represented as a diagram is shown below.

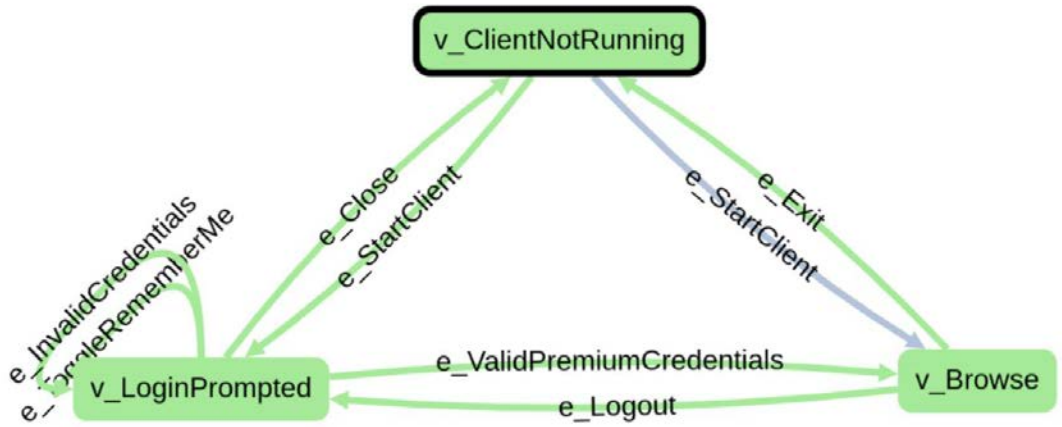

Figure 2: Structure of Model Based Testing format.

The rectangles in the image represent states or "Vertices" in terms of model-based testing and the arrows that are pointing the flow of these states are known as "Edges". When a website is first opened, it can be considered as a state change as the website is now in a state where it is first opened, this is the initial stage. When any website is clicked or some action is performed that has altered this existing state, it can be considered as an edge.

For an example, when visiting google, we load up the webpage first, this is the initial stage i.e., the first vertex. When we type in a search term and click on the search button, we perform an action which leads to change in the state of the webpage to change from the initial search page to the page where the results are displayed.

The notation of the vertices in the MBT format, i.e., the name of the vertices are always prepended with lower case 'v' and the name of the edges are always prepended with lower case 'e'.

The MBT tool which will replay the whole recorded session is called as the "Graph Walker Tool" which will take in the JSON file containing MBT graph. The order of the execution is decided by the generator.

The default generator is the random generator with a test case coverage of 100% and this can also be customized to a specific order and the desired coverage needed.

SPECIFICATIONS

The specification of the tool is as follows. The tool must receive the click stream data which consists of mouse clicks, text changes and state changes such as button toggles. In order to achieve this, we have to implement a way of receiving events from the browser on the client side and this is possible with Java Script event listeners. This must be implemented using a custom Java Script file that must be included into the source of the website. It will then listen to the changes in states and also the loading of webpages and report them back to the tool.

In order to receive this data from the Java Script that is included into the front end, we also have to implement a server that will receive this data. For this purpose, a REST API server is to be used to receive this data. The controller of this server will also need to process this data and convert into MBT format and store it as a JSON file.

For the purpose of this REST API server, I have chosen to use a Spring REST server which is very easy to configure. The REST API server has also been hosted on the local host just as the SUT has been done.

In a real test environment, the tool can be hosted on a separate machine as it is not necessary for both the SUT and the tool to be on a single machine.

3 DESIGN

SOFTWARE ARCHITECTURE OF MBTMODELGENERATOR

The following diagram is a simple architecture that has been used for the development of this tool.

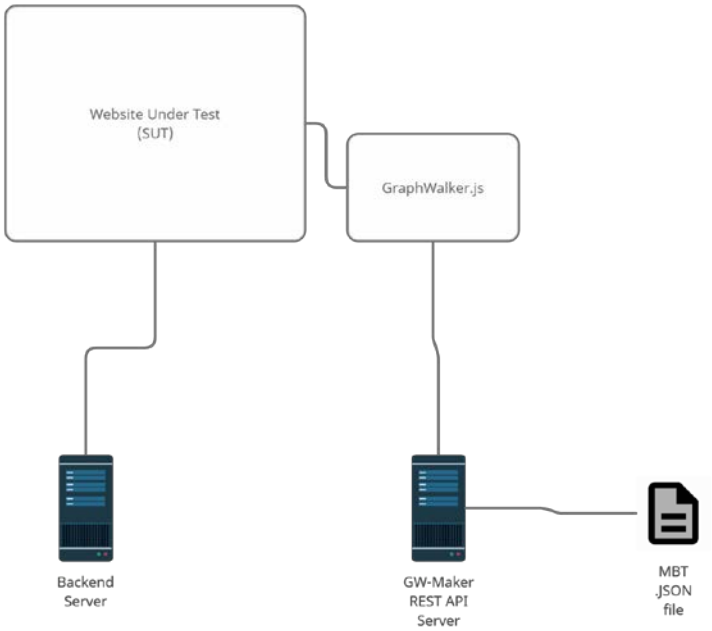

Figure 3: Architectural diagram of GraphWalker-Maker

The Website under test maybe hosted anywhere, either on the localhost or on a separate hosting server. The website will also include the *GraphWalker.js* file that will be used to send the clicks stream events to the *MBTModelGenerator* REST API server. The server will then generate an MBT file in JSON format and saves it to storage.

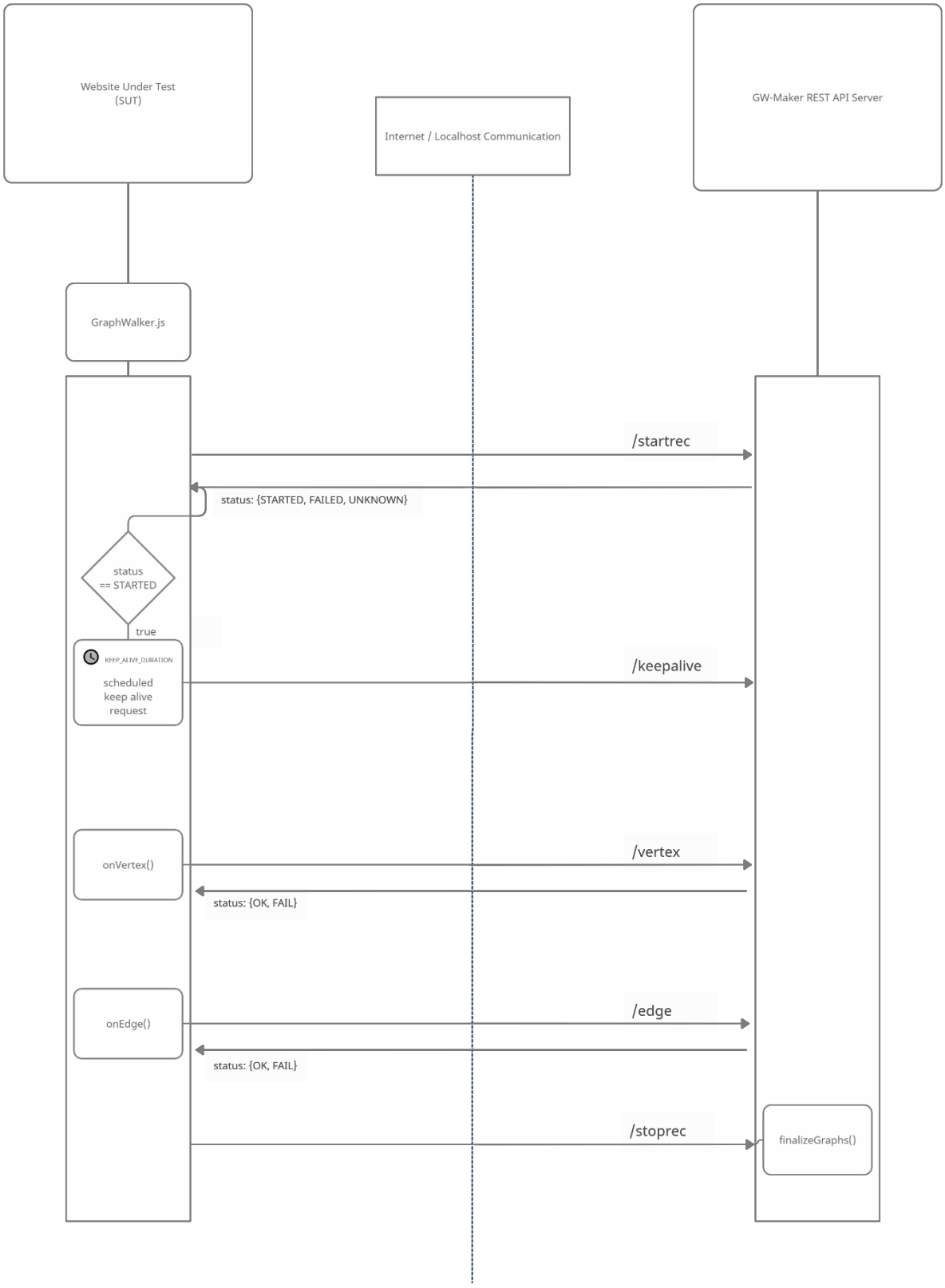

Figure 4: API flow diagram

The above diagram shows the API calls between the website under test and the *MBTModelGenerator* REST API Server. This API calls may occur through the internet or in the localhost.

When the website is first loaded, the *GraphWalker.js* will start up and will send a request to the REST Server to start recording the click streams using the */startrec* endpoint.

This will setup a session and create an empty MBT model where the received vertices and edges will be store into and then later converted into JSON file.

The *MBTModelGenerator* will also have an auto session closure feature where closing the website will automatically stop the recording session and will finalize the MBT graph.

This feature works by short polling the REST API server on */keepalive* end point every KEEP_ALIVE_DURATION number of seconds, this duration has been set to 3.33 seconds.

If the REST Server does not receive any keepalive request in a span on 10 seconds, then the session will be considered finished and the finalisation of the MBT model will start.

When a button is clicked or any input event occurs, it considered an edge and the resultant state or the initial state of this event will be considered a vertex. The edges and vertex are sent to the REST Server using the *GraphWalker.js* script from the endpoints */edge* and */vertex.*The */vertex* endpoint is a POST Request endpoint which will take in a single parameter: name of type String which is the name of the vertex. The */edge* endpoint is also a POST Request endpoint which will take in a single parameter: name of type String which is the name of the edge.

When a capture session is finished, the website may call the finish test function manually to stop recording or the tester may also close the website after the test record and the session will automatically finalize the sessions after 10 seconds.

SEQUENCE DIAGRAMS OF MBTMODELGENERATOR

The following sequence diagram shows the start record endpoint flow. When the REST API server receives the request at this endpoint along with the name of the session, it creates a new *GraphWalker* object from the *GraphWalkerFactory* static class. The *GraphWalker* object is then used to create a new MBT model where the vertices and edges are stored.

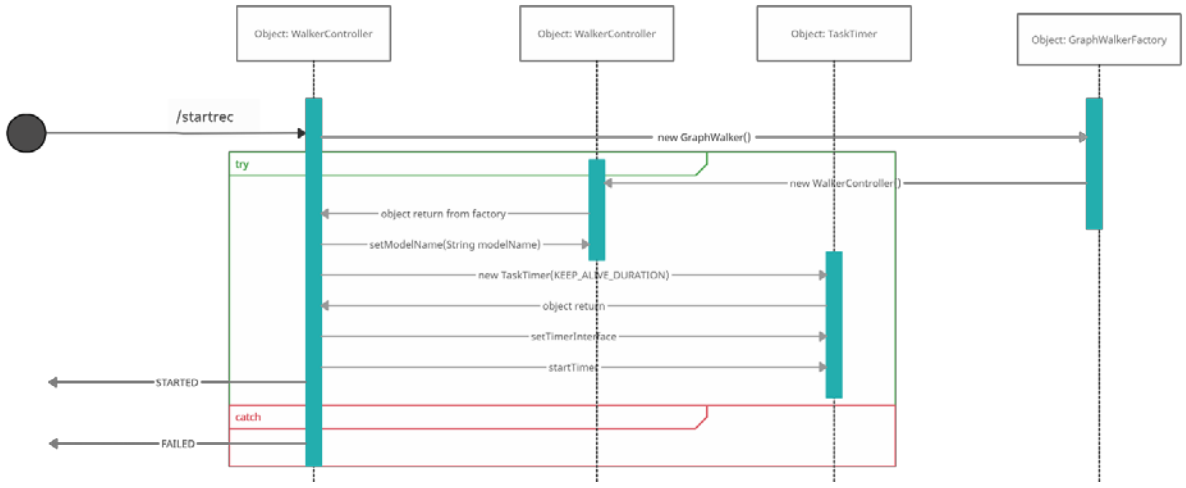

Figure 5: Sequence diagram of start record endpoint.

A *TaskTimer* object is also created which is used to check if the keep alive endpoint has been called in the last 10 seconds. The timer interface is set where the task timer will call back if the keep alive request was not received in time.

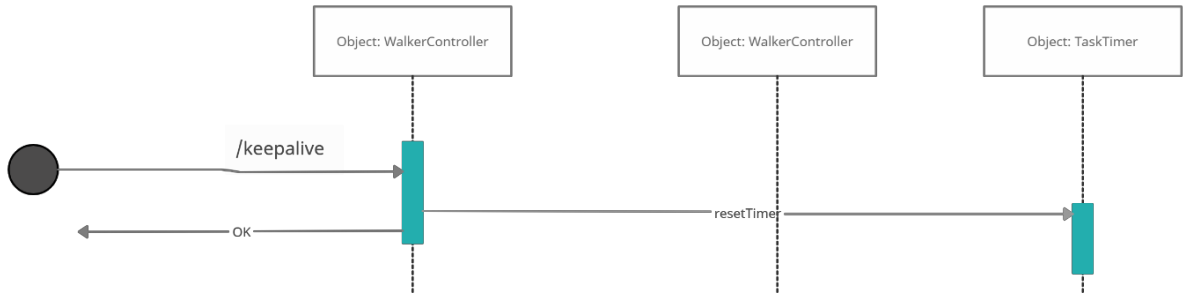

Figure 6: Sequence diagram of keep alive endpoint.

The above sequence diagram shows the timer reset mechanism. Whenever the */keepalive* endpoint is requested, the *TaskTimer* is reset.

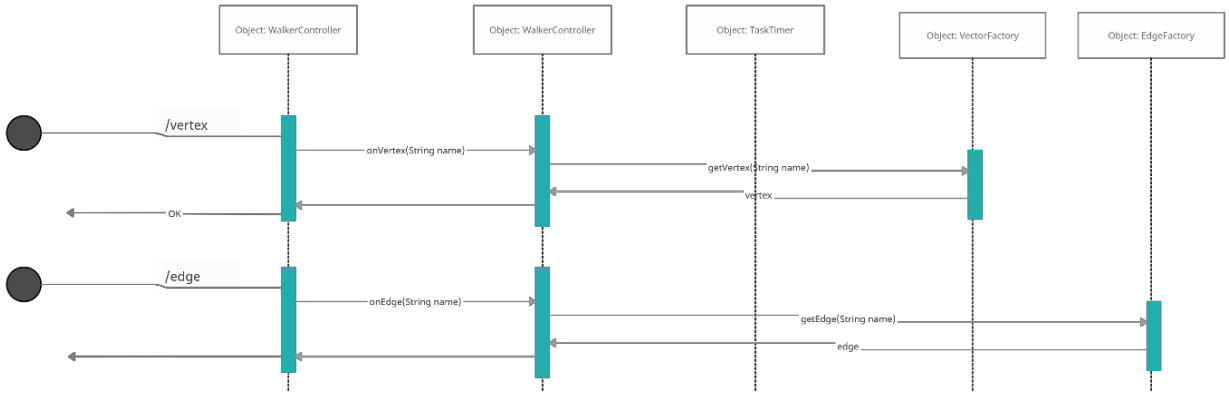

Figure 7: Sequence diagram of vertex and edge endpoint.

The */vertex* and */edge* endpoints are POST requests which take the name of the vertex and edge to be added to the MBT model. The *Vertex* and *Model* objects are created using their respective factory classes.

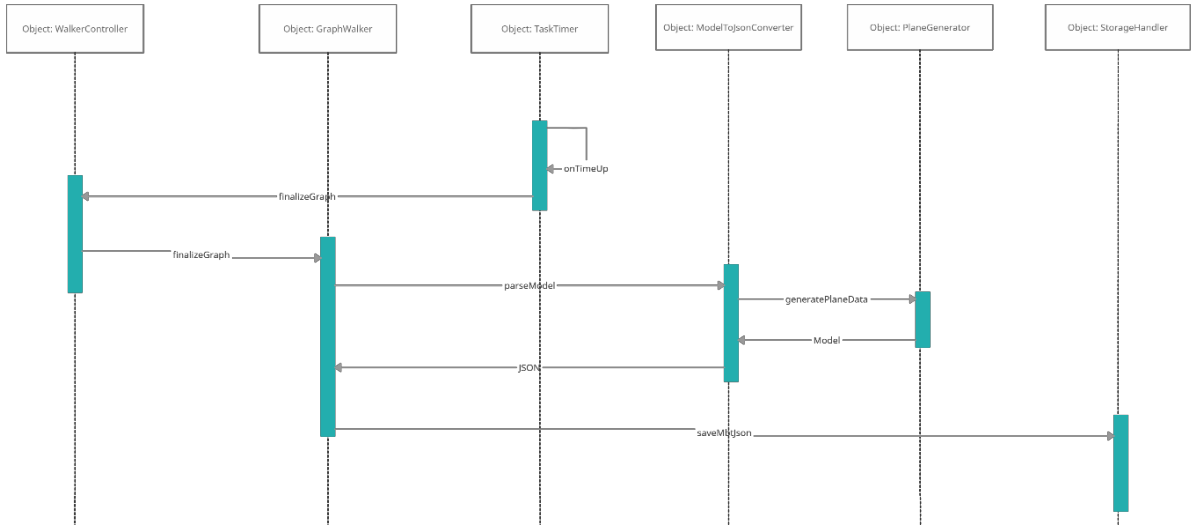

Figure 8: Sequence diagram of automatic session finalisation.

When the website under test does not receive the keep alive request in the duration of 10 seconds, the *TaskTimer* will call the onTimeUp() call back implemented in the *WalkerController* class.

The finalizeGraph method will then be triggered from the callback which will then call the parseModel method in the *ModelToJsonConverter* class to convert the MBT model object into a JSON file.

An MBT model needs co-ordinates for all the vertices and edges, the *PlaneGenerator* class using the generatePlaneData method. The resultant JSON model is the written to a JSON text file into storage using *StorageHandler* class's saveMbtJson method.

4 IMPLEMENTATION

IMPLEMENTATION OF MBTMODELGENERATOR REST API SERVER

The tool developed during the duration of this project has been named "GraphWalker – Maker" which is a REST API server made using spring boot framework. This server will be used as a backend to receive the click-stream data and then generate a MBT graph based on the click-stream sequences.

The server consists of a REST API Controller class named *WalkerController* which consists of several endpoints as shown below:

```
/** This Endpoint must be called before the start of a ClickStream session. ...*/
@PostMapping("/startrec")
public String onStart(@RequestParam("title") String title){...}

/** Call this endpoint to add a new vertex. ...*/
@PostMapping("/vertex")
public String onVertex(@RequestParam("name") String name){...}

/** Call this endpoint to add a new edge. ...*/
@PostMapping("/edge")
public String onEdge(@RequestParam("name") String name){...}

/** This endpoint must be called for every */
@PostMapping("/keepalive")
public String onKeepAlive(HttpServletResponse response){...}

@PostMapping("/stoprec")
public void onStop() { finalizeGraphs(); }
```

Figure 9: Mappings of different endpoints in the REST API Server.

The processing of the click stream has been delegated to another class named *GraphWalker* which has the following function definitions:

```
/** Constructor */
public GraphWalker() {...}

public void setModelName(String name) { model.setName(name); }

public void onVertex(String vertexName){...}

public void onEdge(String edgeName){...}

public void finalizeGraph(){...}
```

Figure 9: Methods used to initialize a session.

The Model Based Test (MBT) format consists of endpoints called *vertices* and actions called *edges*. In the spring boot project, we use factory patterns to generate the *Vertex* and *Edge* objects using the classes *VertexFactory* and *EdgeFactory*.

VertexFactory:

```
/** Check if the vertex already exists in the list of vertices. ...*/
public boolean vertexExists(String vertexName) { return vertices.containsKey(vertexName); }

/** getVertex will create or fetch the vertex associated with the vertexName. ...*/
public XVertex getVertex(String vertexName){...}
```

Figure 10: Factory design pattern to construct new vertices.

EdgeFactory:

```
/** Check if the edge already exists in the list of edges. ...*/
public boolean edgeExists(String edgeName) { return edges.containsKey(edgeName); }

/** getEdge will create or fetch the edge associated with the edgeName. ...*/
public XEdge getEdge(String edgeName){...}

/** This method allows the updated edge to be inserted into the list of edges. ...*/
public void setEdge(XEdge edge){...}
```

Figure 11: Factory design pattern to construct new edges.

The model consisting of the edges and vertices will be converted to a JSON file with MBT format using the *finalizeGraph()* method in the *GraphWalker* class. Another class named *ModelToJSONConverter* will be used to generate a json string from the MBT model object.

```
/**
 * Converts the model object into JSON format.
 *
 * This will later be saved into a json file in storage.
 */
public class ModelToJsonConverter {

    /**
     * Extract all variables from Model object and write them into JSON object.
     *
     * @param model
     *
     * Model object provided by GraphWalker.
     *
     * @return
     *
     * JSON object to be saved.
     *
     */
    public static JSONObject parseModel(Model model){...}

}
```

Figure 12: Class to store the MBT data into JSON format.

The MBT format also must contain geometrical co-ordinates of all the vertices and edges so that they can be visualized. This co-ordinate generation is handler by the *PlaneGenerator* class as shown below:

```java
/**
 * This class handles the placement of the vertices on the 2D X-Y Axis graph.
 *
 * The placement is done on the basis of the number of edges a vertex is connected to.
 *
 * The number of edges is calculated for every vertex and the vertices are then
 * sorted in ascending order based on the number of vertices into a list.
 *
 * Then, the vertices are places on a circular path equidistantly in a zig
 */
public class PlaneGenerator {

    private static int MINIMUM_SEPARATION = 400;

    private static int ROTATION = 90;

    public static Model generatePlaneData(Model model){...}

    private static XVertex populateEdgeCount(Model model, XVertex vertex){...}

    private static double sin(double degrees){...}

    private static double cos(double degrees){...}

}
```

Figure 13: Class to generate co-ordinate data for the vertices in MBT file.

A Unit test has been designed for the test of the *PlaneGenerator* class named *PlaneGeneratorTests* as shown below:

```java
@Test
public void generatePlaneTest(){
    Model model = new Model();
    model.setName("ShoppingCart");
    model.setProperty("generator", new XRPathGenerator().pathJsonName);
    XVertex n2 = vert( name: "Amazon",  id: "n2");
    XVertex n3 = vert( name: "SearchResult",  id: "n3");
    XVertex n4 = vert( name: "BookInformation",  id: "n4");
    XVertex n5 = vert( name: "AddedToCart",  id: "n5");
    XVertex n6 = vert( name: "ShoppingCart",  id: "n6");

    model.addVertex(n2);
    model.addVertex(n3);
    model.addVertex(n4);
    model.addVertex(n5);
    model.addVertex(n6);

    model.addEdge(edge( name: "SearchBook",  id: "e10", n4, n3));
    model.addEdge(edge( name: "EnterBaseURL",  id: "c2a189b6-bd93-4fa8-a32a-c5d0aafe4a0a", n2, n2));
    model.addEdge(edge( name: "SearchBook",  id: "e2", n2, n3));
    model.addEdge(edge( name: "ClickBook",  id: "e3", n3, n4));
    model.addEdge(edge( name: "AddBookToCart",  id: "e4", n4, n5));
    model.addEdge(edge( name: "ShoppingCart",  id: "e5", n5, n6));
    model.addEdge(edge( name: "ShoppingCart",  id: "e6", n3, n6));
    model.addEdge(edge( name: "ShoppingCart",  id: "e7", n4, n6));
    model.addEdge(edge( name: "SearchBook",  id: "e8", n6, n3));
    model.addEdge(edge( name: "SearchBook",  id: "e9", n5, n3));

    JSONObject mbtJson = ModelToJsonConverter.parseModel(
            PlaneGenerator.generatePlaneData(
                    model
            )
    );

    StorageHandler storageHandler = new StorageHandler();
    boolean result = storageHandler.saveMbtJson(mbtJson.toString(), model.getName());

    assertThat(
            result
    ).isEqualTo(true);

}
```

Figure 14: Unit test to verify the co-ordinate generation capability of the PlaneGenerator Class

IMPLEMENTATION OF THE JAVASCRIPT FILE.

The website under test will send the click stream data to this REST API server by reporting the events to the following JavaScript function embedded into the website.

```
var KEEP_ALIVE_DURATION = 3333

var STARTSESS_ENDPOINT = "http://localhost:8496/startrec"
var KEEPALIVE_ENDPOINT = "http://localhost:8496/keepalive"

startSession()

function startSession(){
    console.log("Starting session...")
    var ssr = new XMLHttpRequest()
    var data = new FormData()
    data.append( name: "title", document.title)
    ssr.open( method: "POST", STARTSESS_ENDPOINT)
    ssr.onload = function () {
        console.log("Response: " + this.responseText)
        if(this.responseText == "STARTED"){
            window.setInterval(keepAlive, KEEP_ALIVE_DURATION)
        }
    }
    ssr.send(data)
}

function keepAlive(){
    var kar = new XMLHttpRequest()
    kar.open( method: "POST", KEEPALIVE_ENDPOINT)
    kar.send()
}
```

Figure 15: JavaScript file to communicate with the REST API Server.

The JavaScript function to start a new session has been currently implemented and the functions to send the click stream events of the edges and vertices have to be implemented in the future.

The idea is to call these JS functions when a click or any input events occurs on the user interface which will be send to the REST API server to be stored into an MBT file.

4.1 Implementation of *TaskTimer* class:

The *TaskTimer* class is used with a call back mechanism using the java interface named *ITaskTimer*. It used the *TimerTask* class to create timed interrupts at regular intervals. The startTimer method is used to start the task which is shown below.

```
/**
 * Method used to set up the timer task and start the timer.
 *
 * @throws InvalidAttributeValueException when the timerDelay is not configured or
 * configured to 0 Seconds.
 */
public void startTimer() throws InvalidAttributeValueException {
    if(timerDelay == 0){
        throw new InvalidAttributeValueException("Timer Delay must be greater than 0 Seconds!");
    }
    timer = new Timer( name: "Keep Alive Timer");
    task = new TimerTask() {
        public void run() {
            if(timerInterface != null){
                timerInterface.onTimeUp();
            }
        }
    };
    timer.schedule(task, timerDelay);
}
```

Figure 16: Internal method to start the timer to listen to keep alive requests.

The reset timer method is also implemented so as to restart the timer to check for if keep alive request has been received in the last 10 seconds.

```
/**
 * Reset the Timer.
 */
public boolean resetTimer(){
    if(task == null){
        return false;
    }else {
        try {
            endTimer();
            startTimer();
            return true;
        } catch (InvalidAttributeValueException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Figure 17: Internal method to rest the keep alive listener timer.

5 TESTING

SYSTEM UNDER TEST (SUT) 1: PET CLINIC

The first SUT (System Under Test) that has been chosen is the PetClinic web application[1]. This is the default test website made using Spring Boot framework in Java. It consists of several testable endpoints/ services such as finding owners, adding owners, list of veterinarians, assigning veterinarians, index and an error page.

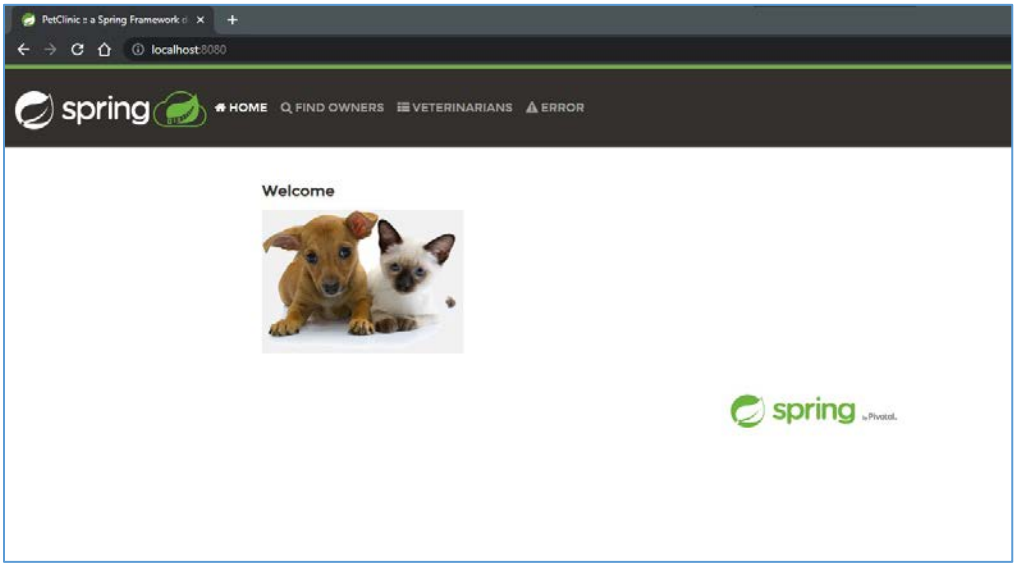

Figure 18: Home screen of Spring Pet Clinic website.

The tester will navigate through this website and the click stream events are collected using the embedded JavaScript file in the website's source. This JavaScript file will communicate with the REST API Server and send all events to it and when the website is finally closed, all the data will be saved into a JSON file with MBT data formatted into it.

[1] github.com/spring-projects/spring-petclinic

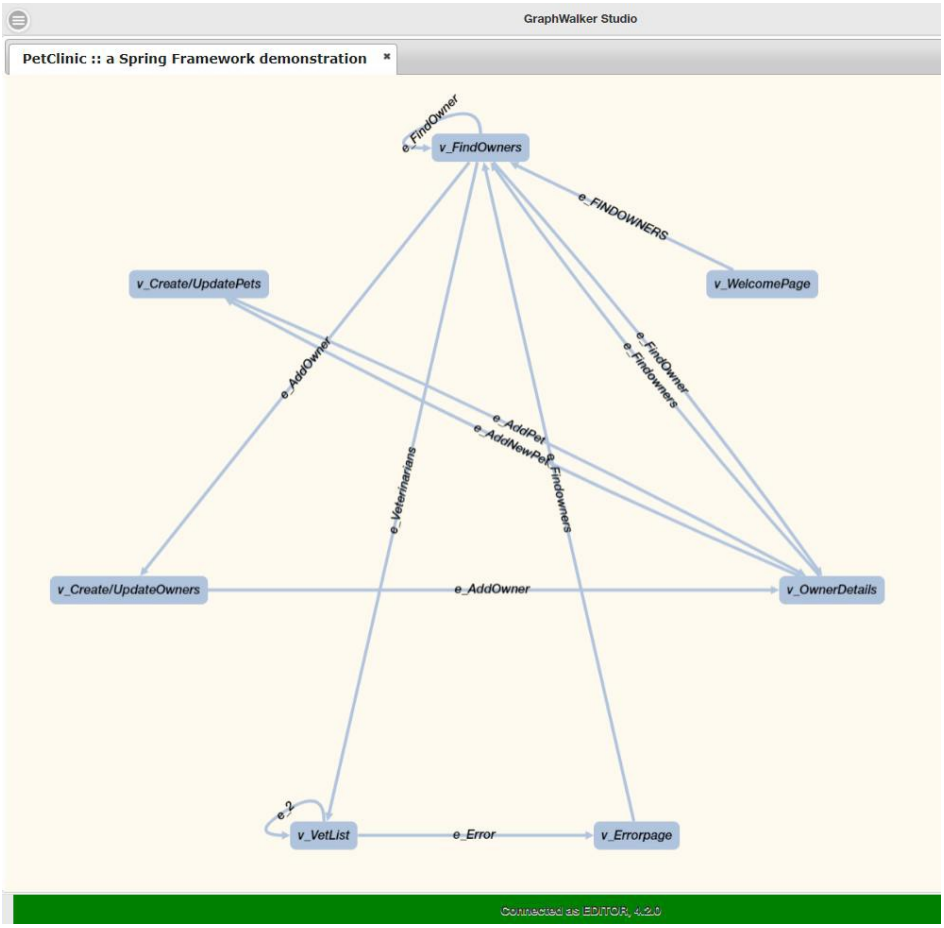

Figure 19: MBT model of the PetClinic website visualized using the GraphWalker tool after test run of MBTModelGenerator tool.

The above diagram shows the various vertices and edges that have been recorded into the MBT .JSON file. The initial state of the website is the *v_WelcomePage* as there is no other edge preceding it.

Several clicks follow it that make up the various edges and the pages that are being loaded as the actions are being shown as the arrows pointing from one vertex to another. When the tester clicks the "FindOwners" button on the top of the website, an edge named *e_FINDOWNERS* is being sent from the JavaScript to the REST API Server. In this similar way, all other clickstream data is being sent from the client to the server.

Upon the closure of the website, the MBTModelGenerator will not receive keep alive requests anymore and after a time out of 10 seconds, the session will be finalized and the .JSON file will be store to the storage after the calculation of the co-ordinates is complete.

In the MBT graph, we can observe that the vertices (the rectangles) have an angular separation, they are laid around a circle and this calculation is performed by the *Planegenerator* class.

SUT2: TASK MANAGER

The task manager website is a software used to create and assign tasks for different users as well as mark them complete. This website has been designed with multi user facility in mind where the manager has the permission to create and assign any task to any other user but the other individual users can only edit their tasks and mark them complete but not fiddle with other user's tasks. This is a fairly complicated SUT and we I have embedded the JavaScript into the source of this website which is based on Spring framework.

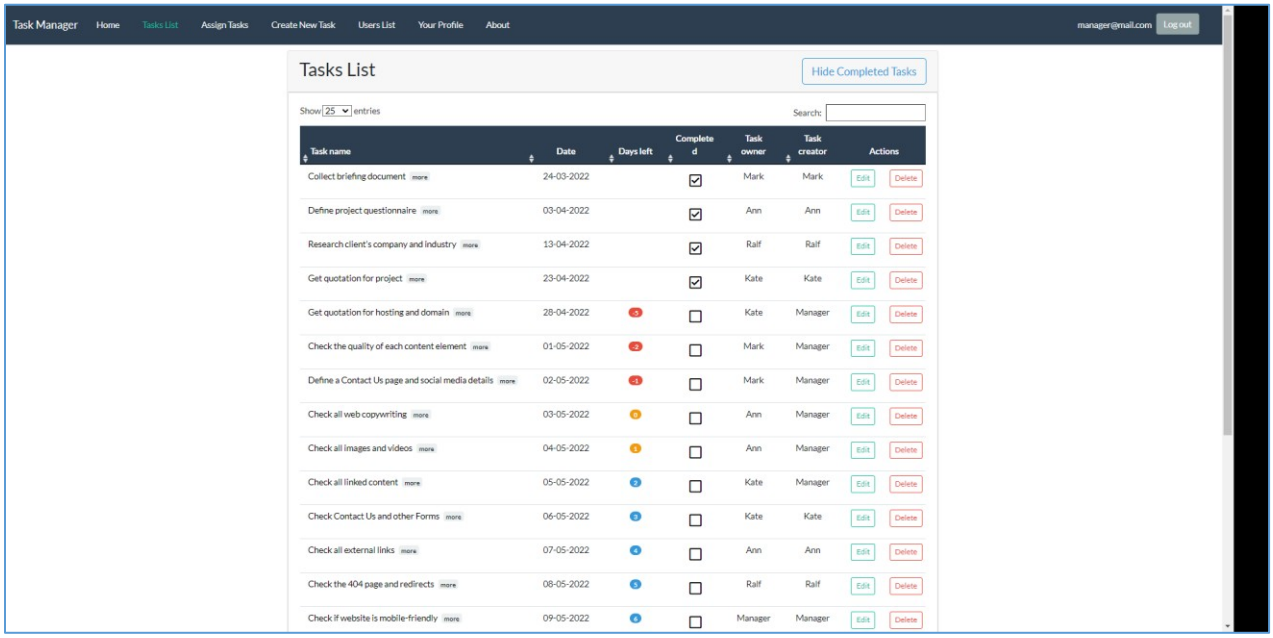

Figure 20: Home screen of Task Manager Website.

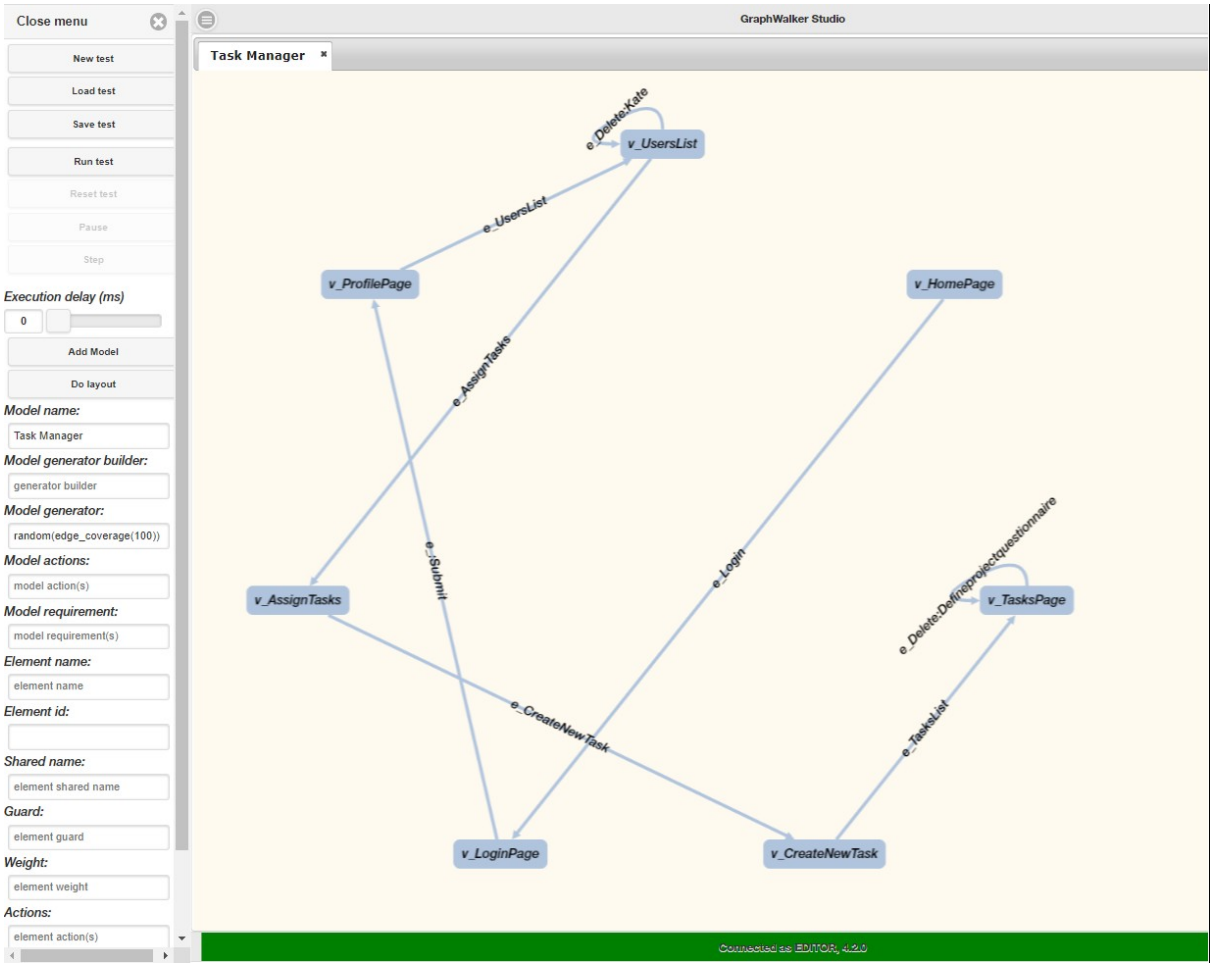

Figure 21: Model based test diagram of the Task Manager website visualized using the GraphWalker tool after test run of MBTModelGenerator tool.

6 SYSTEM EVALUATION

The MBTModelGenerator REST API server has been successfully built and the *GraphWalker.js* needs to be updated the functions to send the vertex and edge events to the server from the website under test. The generation of MBT file from the clickstream events is tested and is working perfectly.

The methodology of evaluation is the usage of Junit tests to check for the correctness of the output of the MBT file and also visual inspection done using the GraphWalker Studio where the verification of the co-ordinate generation is done.

Both of the test websites i.e., Pet Clinic and Task Manager have been tested thoroughly and have shown click stream output for most of the interactions that have been reflected in the MBT graphs.

APPENDICES

APPENDIX A: RUNNING AND USING GRAPHWALKER TOOL

1. The MBTModelGenerator tool must be first started on the localhost.
2. Start the Website under test.
3. The tester must go through the website and test every part of the website manually, this sequence of actions will generate a click stream data which will then store as a MBT JSON file and then can be replayed using GraphWalker player tool.
4. To repay the test of the website, open the GraphWalker player tool, load the json file and click start and the GraphWalker tool will automatically start the tests.

Note: The tester may need to manually add the click Java Scripts for the GraphWalker to perform the actions on the elements of the website.