Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Model-based test generation

MSc Project Laboratory 2

Author
Gergő Ecsedi

Advisor
dr. Zoltán Micskei

March 5, 2017

Contents

Chapter 1

Model-based testing

I have discovered two articles about Model-based testing, which is a Model-based testing taxonomy [1] and an ISTQB Syllabus [2]. In this section, I want to summarize the main ideas from them.

1.1 Test generation introduction

Testing aims at showing that our implemented software and hardware system is suited for our needs. To do that, we first must specify clear goals for our testing. With testing we want to detect system failures and the differences between the expected output and the real implementation's output.

Model-based testing (MBT) is a way to clarify these differences for our system under test (SUT). This test approach uses a model that encodes the intended behaviour of the SUT and possibly the behaviour of its environment. This model should be simple, easy to understand according to the SUT complexity, and easy to check, modify and maintain. The idea of using test models is to avoid the complexity of hand-written tests, which are hard to design, maintain and write. Consequently the model must contain detailed information about the automatic generated tests.

MBT is basically impacts the whole test process, but does not solve everything. Any change in the requirements or in the MBT model propagates to regenerate all the tests and review the correctness, if necessary.

To fit the MBT into the developing process we must consider what inputs and outputs should be given.

Input artefacts:

- Test strategy
- The test basis including requirements and other test targets, test conditions, oral information and existing design or models
- Incident and defect reports, test logs and test execution logs from previous test execution activities
- Method and process guidelines, tool documents

Output artefacts include different kinds of test ware, such as:

- MBT models
- Parts of the test plan (features to be tested, test environment), test schedule, test metrics
- Test scenarios, test suites, test execution schedules, test design specifications
- Test cases, test procedure specifications, test data, test scripts, test adaptation layer (specifications and code)
- Bidirectional traceability matrix between generated tests and the test basis, especially requirements, and defect reports

1.2 General model-based testing process

In this section I want to describe the general process of the model-based testing in 5 steps (In figure: 1.1.), which was described in the article: *A taxonomy of model-based testing* [1].

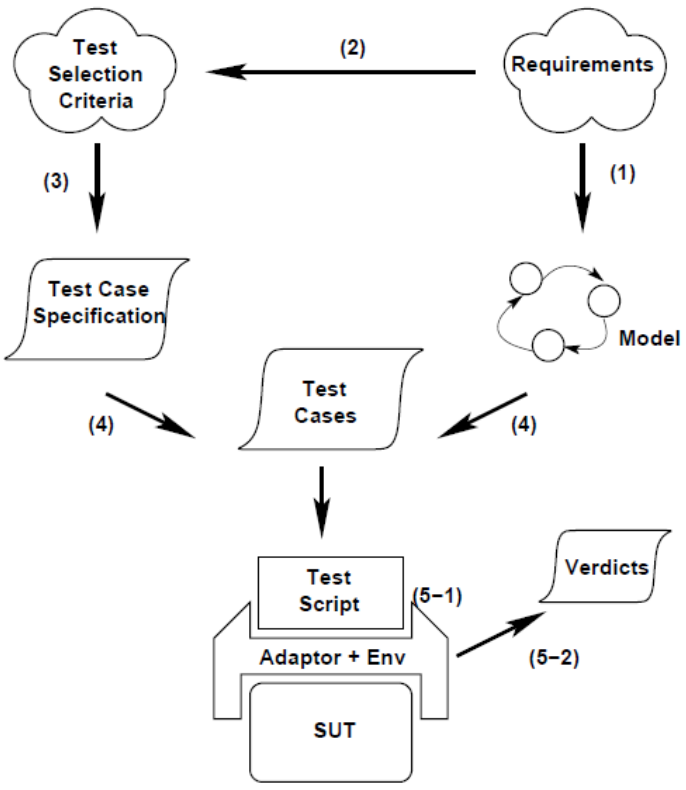

Figure 1.1. MBT process [1]

Step 1. Our first task is to create a model of our System Under Test (SUT), which is implemented by considering the system requirements and specification documents. Our model could have an abstract level implementation, so we can forsake some functionality or certain quality-of-service attributes.

Step 2. In this step we define test selection criteria, which means that we define what is the goal of our tests. The best test is to detect all system's failures and give a helpful identifying what caused the failure. In general, test selection criteria can concentrate to functionality, structure of the model or well-defined set of faults, consequently it is a subset of behaviours of the model.

Step 3. The previously defined selection criteria then transformed into test case specifications. These statements formalise the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite.

Step 4. Test suites are generated from the model, which satisfies the test case specification (but that aggregation can be empty too). The generator pick randomly a test case from the generated set of test cases.

Step 5-1. Because of the abstraction of the model, each test case input and output concretisation handled by an abstraction layer called the adaptor. The executing done by test script applying the considered input and output to the SUT. The adapter and the script is not entirely separated from each other.

Step 5-2. The adapter creates a verdict, which is the result of the comparsion. This can be *passed*, if the expected and the actual output conform. Otherwise the result can be *failed*, which means that the expected output does not match with the actual output or *inconclusive* meaning that there is no decision yet.

1.3 Model-based test generation dimensions

Subject of MBT models Basically we can set up two models for testing purposes, one for our SUT and the other for the environment of our SUT. The first one is encodes the intended behaviour of our system (as an oracle), the environment model is used to restrict the possible inputs to the SUT model (as a test selection criterion). Furthermore while creating a model consider the abstraction level of that. This can be a functional aspect (with limited functionality of the SUT), a data abstraction (expected input or/and output possibility restriction), a communication abstraction (mostly in protocol testing) and quality-of-service abstraction (like security, memory consumption).

Redundancy According to the SUT size, we can have multiple models for test suit. The models can have different aspect or different abstraction for more successful testing.

Quality Characteristic Model quality directly affects the generated test output. MBT tools may check the syntax (model is consistent with the formal rules) and at least partly, the semantic of the model (the content of the model is correct). Reviews check semantic and pragmatic quality (model is proper to test scenario and test generation).

Test Selection Criteria From the same model various test suites can be generated. Below test selection criteria will be described, which can help the tester to specify the right goal of the targeted tests.

The coverage items may be:

- **Requirements linked** to MBT model, so full requirement sheet corresponds to the test cases.
- **MBT model elements** set coverage items to test cases like states, transition and decision in state diagrams.

- **Data-related test selection criteria** is related to test design techniques and may include heuristics such as pairwise test case generation.

Test Generation Technology This have the most biggest influence to the test generation results. Test cases can be generated by random path generation algorithm, dedicated search-based algorithms, model-checking (show a counterexample), symbolic execution (to specific input, which part of the SUT executed) or deductive theorem proving (prove a statement).

Test execution MBT generated test cases can be executed by manually or automatically. For manual execution the generated tests must be usable for manual test running. For automated test execution, test cases must be generated in a form that is executable. To test the SUT from abstract tests an adaptation layer code is needed to bridge the abstraction gap. This adaptation layer can be avoided by automated test scripts.

On-line or Off-line test generation This approach is rather a technical detail of the test generation. With on-line generation, we can manipulate the SUT and test cases while executing them, which means parallel test case generation and execution. Consequently off-line generation is the idea that we create test cases before they are run.

1.4 Model-based test generation tools

In this section I want to describe the Graphwalker (1.4.1) and the PyModel tool (1.4.2), which I have discovered in this semester.

1.4.1 Graphwalker

GraphWalker is an open source model-based testing tool for test automation. It's designed to make it easy to create your tests using directed graphs. The tool generates test paths from these given graphs, which could be connected to each other. Each graph will have it's own set of generator(s) and stop condition(s). An edge in the directed graph represent an action in the system, consequently a vertex means a verification state, where we can check assertions in code. A path is used to call the corresponding methods or functions of your SUT (system under test) by the adapter layer.

The test selection is implemented by an expression, which have the following template: generator(stop_condition_type(condition))). This describes how to cover (random, a_star, shortest_all_paths) and what to cover (requirement, edge, vertex, time and their variations).

There are two ways to generate tests by GraphWalker:

- Offline: The path generation from the graph is done once (typically with command line), and these tests needs to be stored. A test automation system handle the tests.
- Online: The path generation from the graph is created during the execution of the tests, run-time. If you have java coded SUT, it is pretty easy to add annotations to SUT and connect that to the generated paths. (command for Maven: *mvn graphwalker:test*)

For test execution an interface from the models is created by *graphwalker:generate-sources* command. Our job is to implement these interfaces and call the proper SUT functionality for the given edges and vertexes.

1.4.2 Pymodel

The second model-based test generation tool, which I have known is Pymodel. This is an open-source framework implemented in Python.

Basically it has 3 parts/programs

- pma: PyModel analyzer: it's parameter gets models, which can be a list of one or more modul names. Each model contains a model (model program, FSM or test suite). This part generates FSM, the explored states and other result of the analysis.
- pmg: PyModel graphics: it's argument is an FSM (created by pma), and the program generates a file of commands in dot graph-drawing language (can be viewed by Graphviz).
- pmt: PyModel tester: it's input parameter is a collection of models like in pma. The tester generates traces by executing the model. This programs offers several possibilities like view the traces, offline and online test generator execution (traces can be saved).

The tool serves us one more program to invoke the 3 other parts with one single command, called pmv (PyModel viewer).

Chapter 2

Case Study

In this chapter I want to demonstrate the functionalities of Model-based test generation. For this I have created sample system, which models a garage gate. First I describe the common functionality of this system, then I set-up the system requirements and finally I introduce the implementation and design decisions of the system.

2.1 System introduction

This garage gate software system consists of 5 different units: two gates, a lamp, a control switch, a control logic and a motion sensor. With the control switch we can open or close the gate. While closing the gate, if someone or something got between the gates, the movement stops. The motion sensor detects this interruption or free status between the gates. Before restarting the opening / closing action, the control unit waits 5 seconds for the lighting warning.

2.2 Software Requirements

Remote Controller

[REQ-01-1] The controller should have an open and a close gate action. [REQ-01-2] One action must be completely finished to start a new action. [REQ-01-3] When the battery is low, the remote controller should warn this to the control logic.

Lamp

[REQ-02-1] The lamp lighting frequency should be 1 second. [REQ-02-2] One lighting section is 6 seconds long.

Motion Sensor

[REQ-03-1] The sensor must detect if anything is between the two pillars of the gate. [REQ-03-2] If the sensor detect blocking thing in the gate, a *blocking* signal must be sent immediately to the *Control Logic*. [REQ-03-3] If the sensor can not detect anything between the two gates, a *free* signal must be sent to the *Control Logic* [REQ-03-4] When the sensor detects free status after the blocking status, there must be at least 3 seconds until the first *free* signal can be sent to the *Control Logic*.

Control Logic

[REQ-04-1] The *Control Logic* can get new actions from the *Remote Controller*, while no action is being processed. [REQ-04-2] While the gates are moving (by an action from the *Remote Controller*), the *Control Logic* should stop the movement, if it gets a *blocking* sign from the *Motion Sensor*. Consequently the *Control Logic* should continue the movement, it it gets a *free* sign from the *Motion Sensor*. [REQ-04-3] The *Control Logic* can receive the signals from the *Motion Sensor*, *Lamp*, *Remote Controller* [REQ-04-4] While the *Lamp* is lighting the gates should not move.

2.3 System implementation

First I want to model the physical representation of the sample on the 2.1. figure.

The system consists 5 elements:

- Lamp: This component is represents the lamp itself.
- MotionSensor: These sensors are on the two pillars of the gate, and detect if something is in between of these sensors.
- RemoteControl: This block is for the remote controller physical realization.
- Motor: This block can move the gates to opened and closed state.
- Controller: This involves the safety logic and the gives control messages to the blocks.

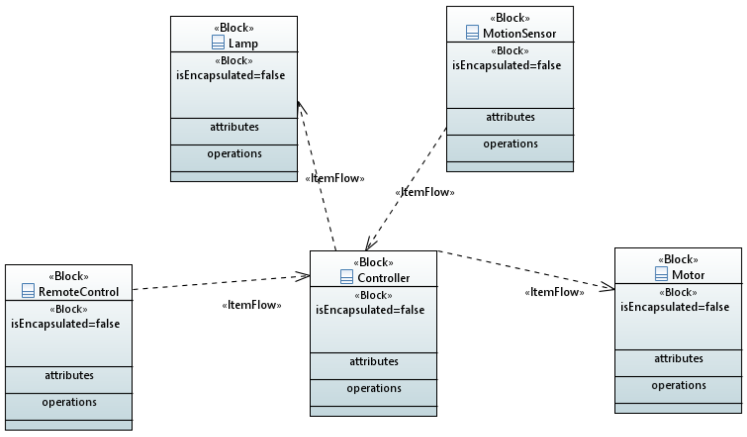

Figure 2.1. Garage system physical components

The 2.2. figure introduce the communication between the above listed blocks.

A garage gate fundamentally have 2 main states, the *Opened* and *Closed* states, which is shown below on 2.3. figure, with orange colours. First of all we can start from the *Closed* state, where we can open the gate with an 'open' command. This command sets the state machine in an *Opening* state. While opening the gate, somebody or something can move into the way, so this becomes *Block Opening*. The gate is opening, if the blocking stops. After the *Opening* phase succeeded the gate is *Opened*. In this state we can 'close' the gate with a simple command, and the state machine goes to the *Closing* state. There could be also a blocking action, which stops the closing movement. From this state the gate is starting the closing movement again after a few seconds *Lighting*. When the closing action finished the gate is *Closed*.

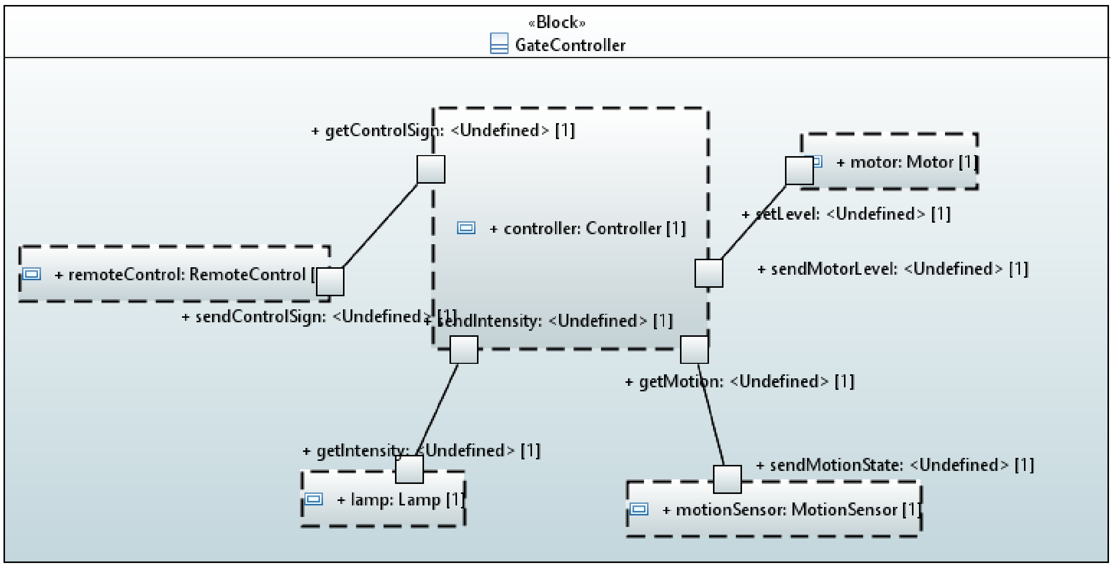
«Block»
GateController
+ getControlSign: <Undefined> [1]
+ controller: Controller [1]
+ motor: Motor [1]
+ setLevel: <Undefined> [1]
+ sendMotorLevel: <Undefined> [1]
+ remoteControl: RemoteControl [1]
+ sendControlSign: <Undefined> [1]
+ sendIntensity: <Undefined> [1]
+ getMotion: <Undefined> [1]
+ getIntensity: <Undefined> [1]
+ sendMotionState: <Undefined> [1]
+ lamp: Lamp [1]
+ motionSensor: MotionSensor [1]

Figure 2.2. Gate controller internal block diagram

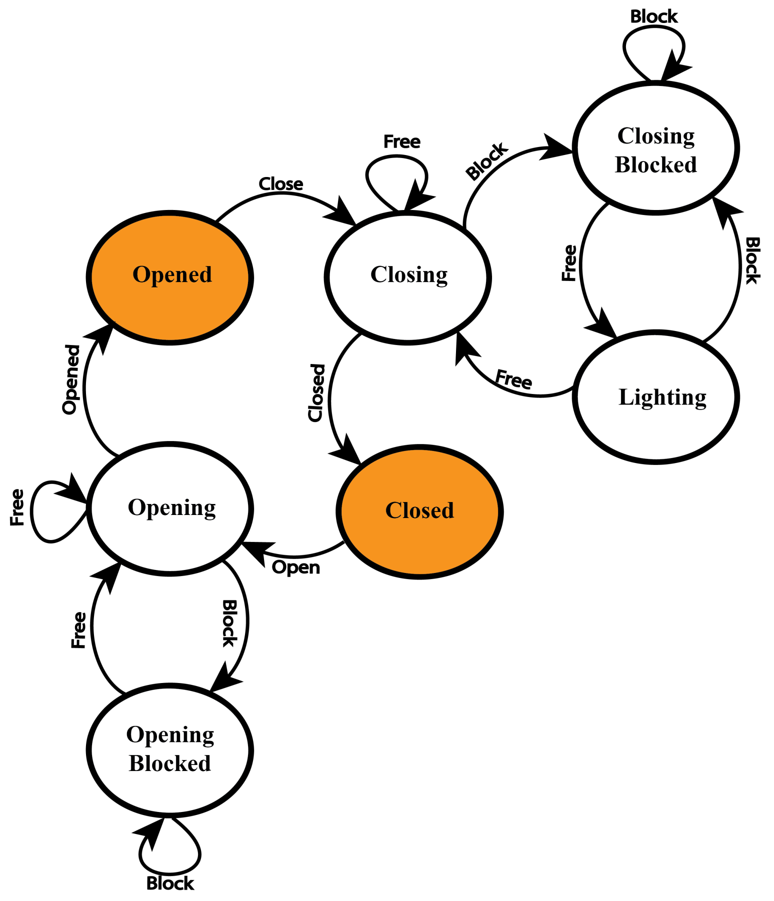

Figure 2.3. Garage gate state machine diagram

Testing goals The tests should achieve all the possible states of the SUT, because we want to detect the behaviour of the SUT in all use cases.

Chapter 3

GraphWalker

3.1 Model for test cases

In this section I will describe the directed graphs for test generation with GraphWalker tool. These models were created in Yed [3].

The first model represents a bit complex state machine for the *Gate Controller* software component. In this model at the *Lighting* state, we concatenate two models with the *SHARED:LIGHTING* keyword in the vertex. So if we get to that state, we propagate our control to the second model, which is shown on the 3.2. figure.

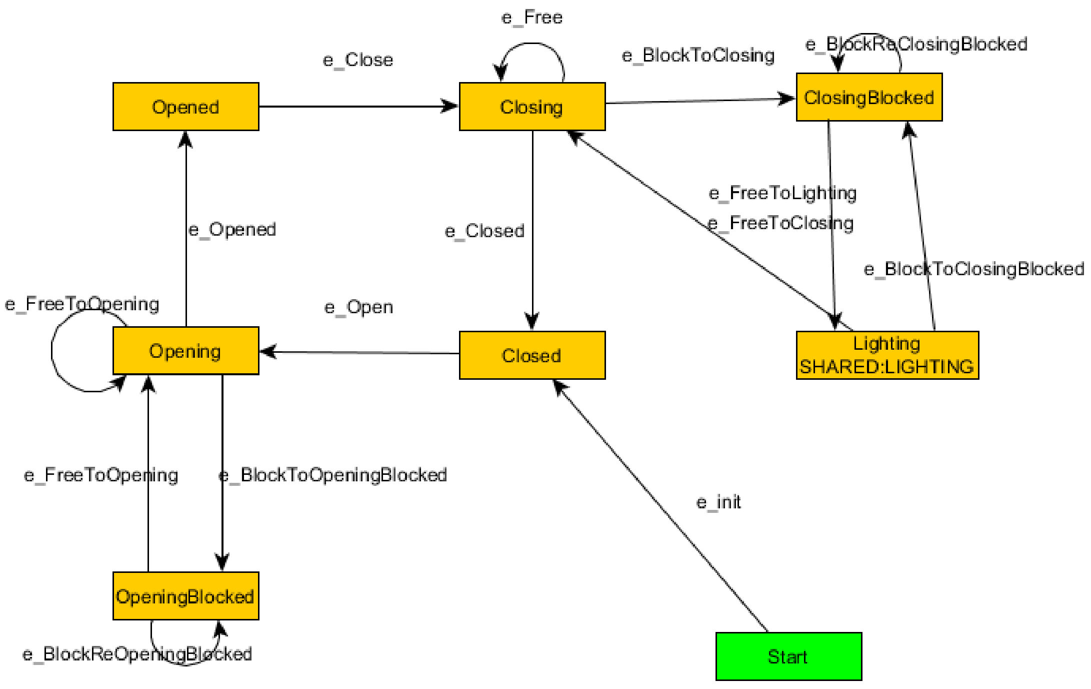

Figure 3.1. Gate Controller model

The control comes to that vertex, which have the some keyword. This model is simple because the lamp can be in a *Lighting* state and can be *Off*.

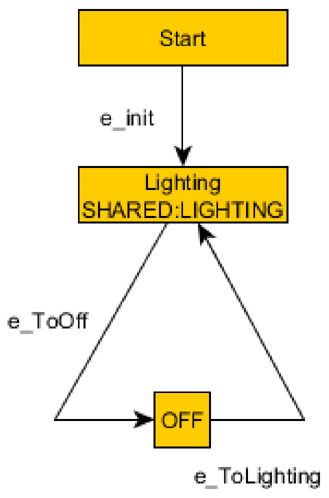

Figure 3.2. Lamp model

3.2 Test adaptation code and test scenario

For the Gate Controller model I have defined the following test scenarios by the GraphWalker command:

```
@GraphWalker(start = "e_init", value = "random(vertex_coverage(100))")
```

The test's goal is to achieve the 100% vertex coverage by random algorithm, because the software system must be able to function in every possible states.

For each vertex and edge in the model, GraphWalker have generated a method. In the test adaptation phase I have implemented these methods to call the SUT functions, for example:

```
//Create a new instance from the SUT
public GateModelTest() {
        super();
        gate = new GarageGate();
}
//implement the e\_Open edge
@Override
public void e\_Open() {
        gate.Open();
}
```

3.3 Test execution results

I have run tests on the gateModel with 100% vertex coverage and as we can see on the next result snippet, the tests reached all vertexes (*"totalNumberOfUnvisitedVertices": 0*).

```
[INFO] Result :
[INFO] {
"totalFailedNumberOfModels": 0,
"totalNotExecutedNumberOfModels": 0,
"totalNumberOfUnvisitedVertices": 0,
"verticesNotVisited": [],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 12,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [
{
"modelName": "GateModel",
"edgeId": "e2",
"edgeName": "e_Free"
},
{
"modelName": "GateModel",
"edgeId": "e6",
"edgeName": "e_BlockToClosingBlocked"
},
{
"modelName": "GateModel",
"edgeId": "e7",
"edgeName": "e_FreeToClosing"
}
],
"vertexCoverage": 100,
"totalNumberOfEdges": 15,
"totalNumberOfVisitedVertices": 7,
"edgeCoverage": 80,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 3
}
```

The result also contains information about which edges was not visited: e_Free, e_BlockToClosingBlocked, e_FreeToClosing. To make sure that these are not inaccessible edges I have run a test with 80% edge_coverage function. The result:

```
[INFO] Result :
[INFO] {
"totalFailedNumberOfModels": 0,
"totalNotExecutedNumberOfModels": 0,
"totalNumberOfUnvisitedVertices": 1,
"verticesNotVisited": [{
"modelName": "GateModel",
"vertexName": "OpeningBlocked",
"vertexId": "n7"
}],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 12,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [
{
"modelName": "GateModel",
"edgeId": "e11",
"edgeName": "e_BlockToOpeningBlocked"
},
{
"modelName": "GateModel",
"edgeId": "e12",
"edgeName": "e_FreeToOpening"
},
{
"modelName": "GateModel",
"edgeId": "e13",
"edgeName": "e_BlockReOpeningBlocked"
}
],
"vertexCoverage": 85,
"totalNumberOfEdges": 15,
"totalNumberOfVisitedVertices": 6,
"edgeCoverage": 80,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 3
}
```

As we can see from the result there are 3 unvisited edges out of 15, which is 80% coverage.

3.4 PyModel implementation

Bibliography

[1] Bruno Legeard Mark Utting, Alexander Pretschner. A taxonomy of model-based testing. 34(1):1–19, 04 2006.

[2] Bruno Legeard Armin Metzger Natasa Micuda Thomas Mueller Stephan Schulz Model-Based tester Working Group: Stephan Christmann (chair), Anne Kramer. Istqb foundation level certified model-based tester. 34:8–35, 2006.

[3] yEd graph editor. xworks the diagramming company (2016. december 13.). `https://www.yworks.com/products/yed`.