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 1

Author
Gergő Ecsedi

Advisor
dr. Zoltán Micskei

December 14, 2016

Contents

Kivonat

Tesztgenerálásra alkalmas eszközök megismerése.

Abstract

In this semester we want to have a look at the test-generator idea and tools.

Chapter 1

Introduction

MSc Project Labor 1

Chapter 2

Model-based testing

2.1 Test generation

Testing aims at showing that our system, which is under construction, is suited for our system requirements. With system testing we want to detect system failures and the differences between the expected output and the real implementation.

Model-based testing is way to clarify these differences for our system under test (SUT). This test approach uses a model that encode 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 unstructured tests written be hand. Consequently it must contains detailed information about the automatic generated tests.

MBT is basically a matter of tooling so this impact the whole test process ,but not solves everything. Any change in the requirements or in the MBT model propagates to regenerate all the tests and review the correctness.

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

Input artifacts:

- 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 artifacts include different kinds of testware, 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

2.2 General model-based testing process

In this section I want to describe the general process of the model-based testing in 5 steps (2.1.).

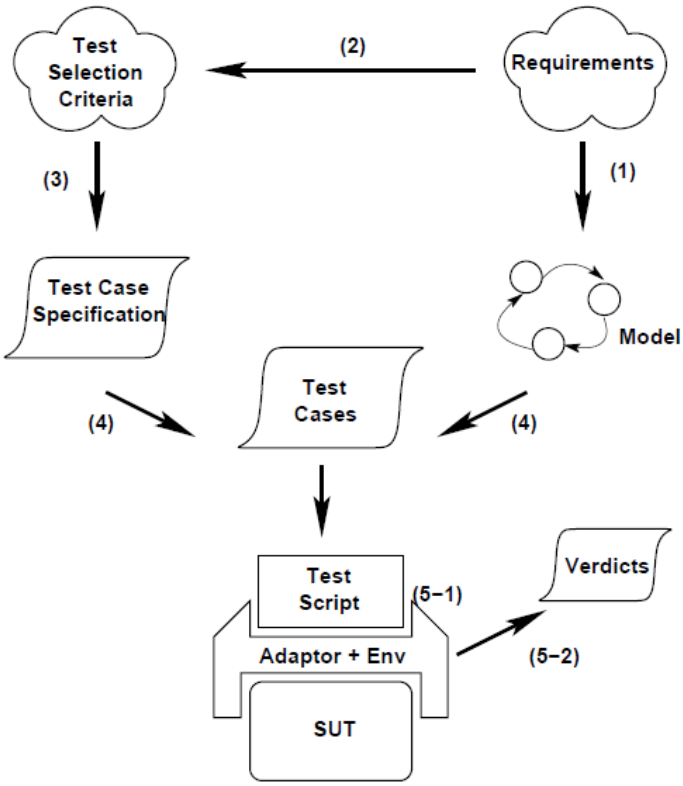

Figure 2.1: MBT process

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

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

Step 3 The previously defined selection criteria then transformed into test case specifications. This specification is a formal aspect of test selection criteria.

Step 4 From the model and the test case specification are completed, a test suite is generated from them. Test case generators pick randomly a test case, which satisfy a test case specification.

Step 5-1 Because of the abstraction of the model, each test case input and output concretisation handled by an abstraction layer called adaptor. The executing done by test script applying the considered input and output to the SUT and build the verdict. 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 *pass*(if the expected and the actual output conform), *fail* (if the expected does not match with the actual output) or *inconclusive* (there is no decision yet).

2.3 Model-based test generation dimensions

2.3.1 Subject of MBT models

To create and effective and maintainable model avoid non-function aspects (like security, timing), further more consider the abstraction level of the model (who will discuss the test design). 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).

Mainly three subjects can be reviewed for the model: **System model**, which is describes the system with class or state diagrams. **Environment model**, which can be deduced from the inputs and the usage of the SUT. This is an opposite aspect then the previous one. **Test model**, which is a model of (one or several) test cases.

2.3.2 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.

2.3.3 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)

2.3.4 Test generation

Selection criteria

From the same model various test suites can be generated. In this section 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** (full requirement sheet corresponds to the test cases), **MBT model elements** (Basically set coverage items to test cases like states, transition and decision in state diagrams). One more aspect is the **Data-related test selection criteria** which is related to test design techniques and may include heuristics such as pairwise test case generation.

Technology

This have the most biggest influence to the test generation results. This technology can be random path generation algorithm or search-based algorithms. In addition this technology could have more logic like model-checking (show a counterexample) and symbolic execution (to specific input, which part of the SUT executed) or deductive theorem proving (prove a statement).

2.3.5 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.

2.3.6 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.

2.4 Model-based test generation tools

What tools i have used, and what they are capable of.

2.5 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 (edge_coverage, vertex_coverage, requirement_coverage, dependency_edge_coverage, time_duration, length, never).

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 graph-walker: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.

2.6 Pymodel

The second Model-based test generation tool, which I have discovered is the 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 b ea list os 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 3

Case Study

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

3.1 System introduction

The system is a garage gate, which consists of two gates, a lamp, a control switch, a control logic and a movement sensor. With the control switch we can open or close the gate. When someone or something got into the gate, while it is closing, the gate stops. Consequently if the gate is free again (nobody has been detected by the movement sensor) it can continue the closing session after some seconds long lamp lighting warning.

3.2 Requirements

Control flow

[REQ-01-1] The control switch should have an open and close button. [REQ-01-2] The signal from the control switch should be reachable up to 200m. [REQ-01-3] Open/Close control action must have ended in Opened/Closed state of the gate.

Lamp

[REQ-02-1] While the lamp is lighting the gate should not move. [REQ-02-2]

Movement 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 Gate Logic. [REQ-03-3] If the sensor does not detect anything in the gate, it must wait for 5 seconds and then send a free signal to the Gate Logic.

Control Logic

[REQ-04-1] The opening and closing movement must be started with a signal from the control switch.

3.3 System realization

The system consists 4 elements, which are shown on the 3.1. figure. The *Gate* element stands for the physical representation of the Garage Gate itself, and can be in *Opened* and *Closed* states. We can open and close the *Gate* with a *Remote Controller*. If we start an action we can not pause or stop that. Nevertheless there is a *Movement Sensor* on the two pillars of the *Gate*, which stops the movement. When the *Gate* is opening and the *Movement Sensor* observes an object between the two pillars, it stops the movement (*Opening Blocked*), until the object is not there any more. In the other case, when the *Gate* is closing and the *Movement Sensor* sign appears, it stops the movement again (*Closing Blocked*). When the object is outside of the scope of the *Movement Sensor* a *Lamp*s on the pillars are *Lighting* for some seconds then the *Gate* is closing again. So the *Lighting* comes only in the closing interruption.

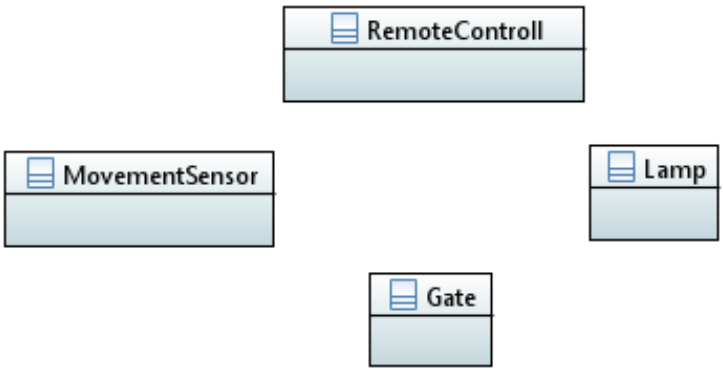

Figure 3.1: Garage gate components

These components can communicate to each other directly. The possible communication messages is shown on the 3.2.

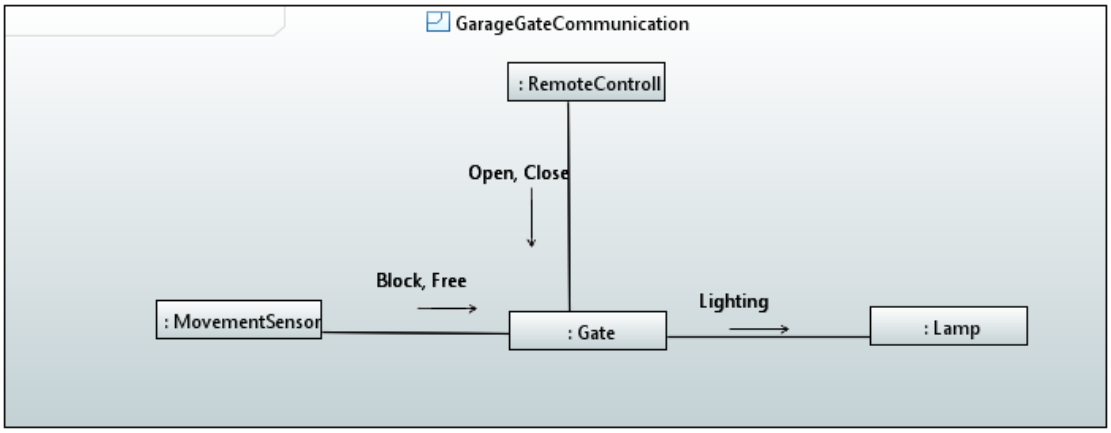

Figure 3.2: Garage gate communication diagram

A garage gate fundamentally have 2 main states, the *Opened* and *Closed* states, which is shown below on 3.3. 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*.

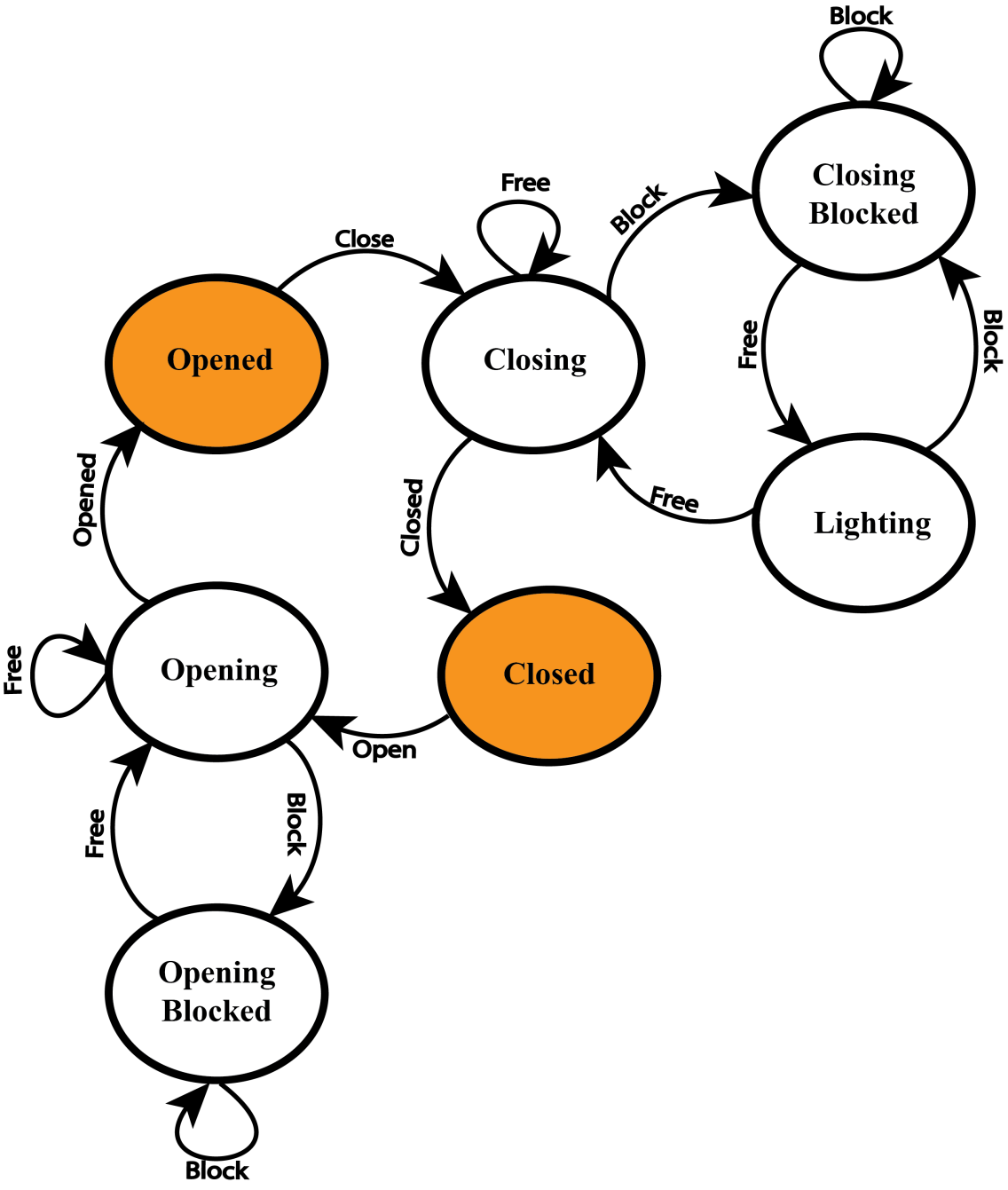

Figure 3.3: Garage gate state machine diagram

Chapter 4

GraphWalker

4.1 GraphWalker

4.2 GraphWalker implementation in Eclipse

We can connect more models with the *SHARED:someName* keyword in a vertex. Consequently I have created a yEd graph model for test scenarios for the garage gate state machine (see 3.3.), and a graph model for the lamp, which called from the previous graph.

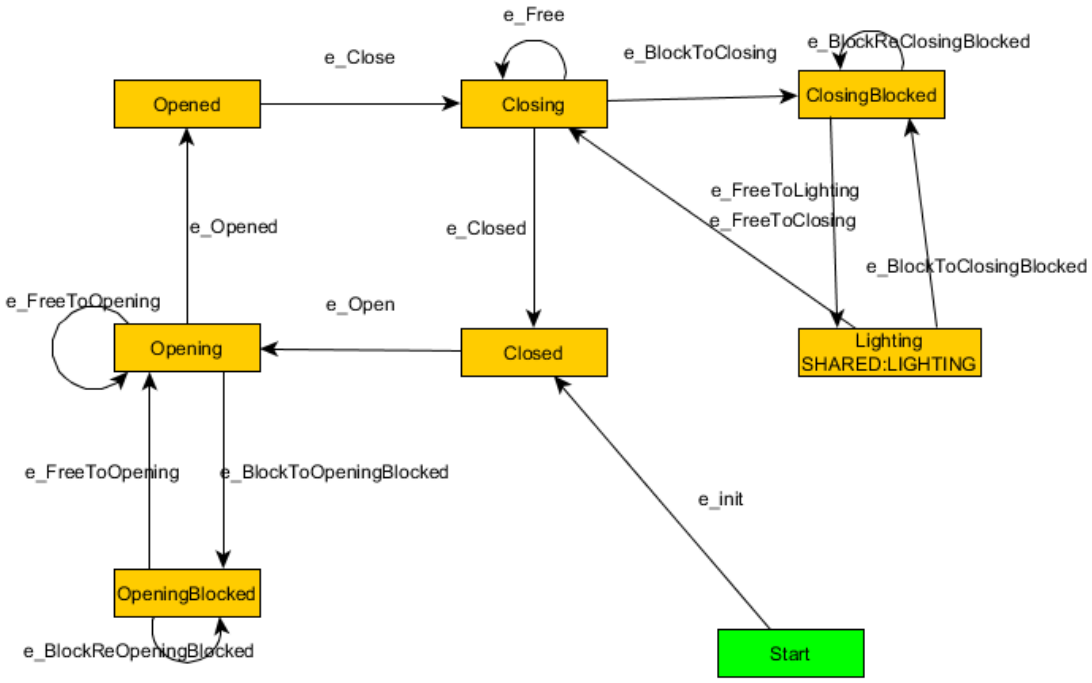

Figure 4.1: Garage gate model with yEd

Generated test with option: @GraphWalker(start = "e_init", value = "random(vertex_coverage(100))"), and the result was: [INFO] Result : [INFO] [INFO] "totalFailedNumberOfModels": 0, "totalNotExecutedNumberOfModels": 0, "totalNumberOfUnvisitedVertices": 0, "verticesNotVisited": [], "totalNumberOfModels": 2, "totalCompletedNumberOfModels": 2, "totalNumberOfVisitedEdges": 14, "totalIncompleteNum-

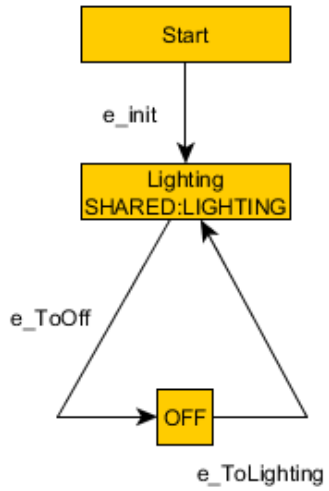

Figure 4.2: Lamp0 model with yEd

berOfModels": 0, "edgesNotVisited": ["modelName": "GateModel", "edgeId": "e0", "edgeName": "e_init" , "modelName": "GateModel", "edgeId": "e3", "edgeName": "e_Close" , "modelName": "GateModel", "edgeId": "e10", "edgeName": "e_FreeToOpening" , "modelName": "GateModel", "edgeId": "e13", "edgeName": "e_BlockReOpeningBlocked"], "vertexCoverage": 100, "totalNumberOfEdges": 18, "totalNumberOfVisitedVertices": 9, "edgeCoverage": 77, "totalNumberOfVertices": 9, "totalNumberOfUnvisitedEdges": 4

4.3 PyModel implementation

Appendix