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

May 23, 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 by examine 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 model can be built in different scenarios depending on our goal of testing.

The idea of using test models is to avoid the complexity of hand-written tests, which are hard to design, maintain and write. The model can have different abstraction levels, but must contain detailed information for the automatic test generation.

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].

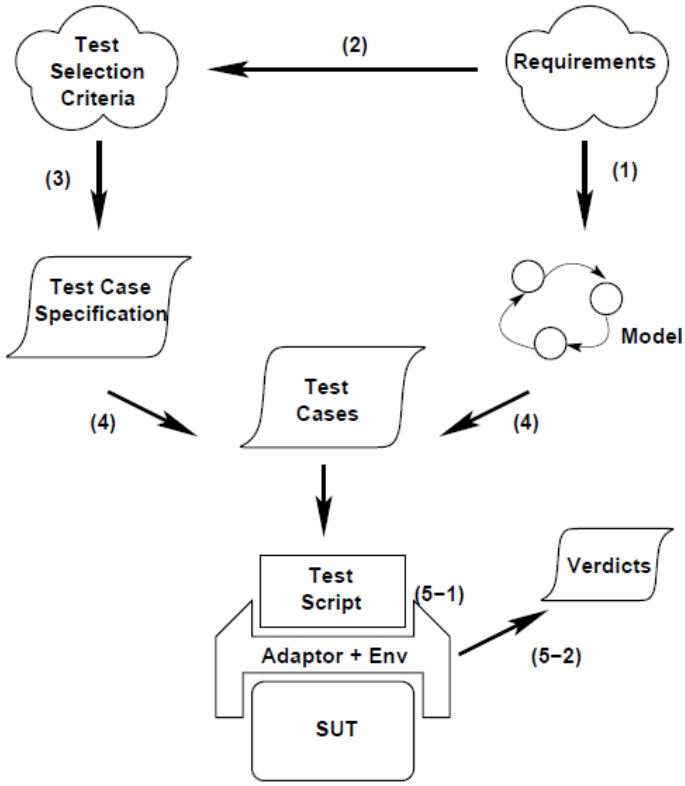

Figure 1.1. MBT process [1]

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 the model's behaviours.

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 adaptor and the script is not entirely separated from each other.

Step 5-2. The adaptor (or the test script) 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 Spec Explorer 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 MBT. It is designed to make directed graph based models easily for testing purposes. The given graphs can be connected to each other, and can have variables too. The tool generates test paths from these given graphs. 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 have assertions in code. A path is used to call the corresponding methods or functions of our 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 Spec Explorer

It is a tool that extends Microsoft Visual Studio for creating models of software behaviour, analysing those models with graphical visualization, checking the validity of those models, and generating test cases from the models. The environment incorporate into our solution in Visual Studio, with a project, that contains a model (adapts the test generation script to the implementation), and a cord file (sets up the SpecExplorer settings). We can create several instances from the model and cord file in the same project.

The complex SpecExplorer environment gives us validation, exploration and test generation opportunities through developing/testing our system. Validation check the consistency of the cord and model programs, then exploration can give graph represented behaviour description about the SUT. In the last steps we can generate test code from the selected machines, which results in standalone unit test cases, which can be run by the other Visual Studio tools. Important information about the selected machines, that these must have TestEnabled and ForExploration flags in the cord files.

Chapter 2

Case Study

In this chapter I want to demonstrate the functionalities of Model-based test generation. For this I have created a 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 for different model-based testing tools.

2.1 System introduction

The user can send action with the control switch to the control logic meaning that open or close the gate. If someone or something suddenly appears in the way of the gate, while it is moving, the movement stops. The motion sensor detects this interruption (the gate is blocked) or free status (the gate has free way). Before restarting the closing action, the control unit waits 5 seconds for the lighting warning. When the gate is finally closed, an additional sensor gives us plausibility for the gate physically closed state.

In this garage gate sample, we can have sensors, which measures the world around the system, and subsystems for realizing some logic in their functionalities.

This garage gate software system consists of 6 different units (see figure 2.1.). These are the following: a gate, a lamp, a remote Controller, a control logic, motion sensor, a motor and a gate closed-state detection sensor.

The communication between the components is based on dedicated signal types like MotorSignal or LampSignal. These types mainly focusing on the control commands for each block.

The system consists 6 elements:

- Lamp: This component is represents the lamp itself, which can be turn off or on.
- Motor: This block can move the gates to opened and closed state.
- MotionSensor: These sensors are on the two pillars of the gate, and detect if something has appeared between them.
- SwitchController: This block is for the remote controller realization. It has an *open* and a *close* button function.
- GateStateSensor: This sensor can detect if the gate is in the final closed or opened state, when physically the movement must be stopped.

- GarageGateLogic: This block implements the safety logic for the gate system, therefore it has all of the necessary information about the components.

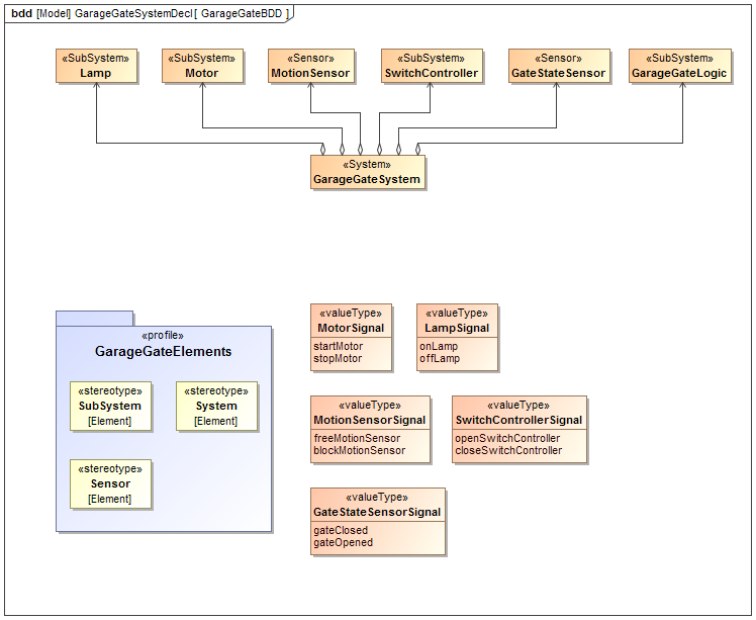

Figure 2.1. Garage system components and signals

2.2 Software Requirements

2.2.1 Lamp

REQ-01-1 The lamp lighting frequency should be 1 second.

REQ-01-2 One lighting section is 5 seconds long.

REQ-01-3 The warning lighting must happen while the gate is closing, and the motion sensor gives free status after a blocked event.

2.2.2 Motor

REQ-02-1 While the motor is running, the gate should get into the opposite state (opened/closed).

REQ-02-2 The motor moving operation can be stopped and resumed by the control logic.

2.2.3 Motion Sensor

REQ-03-1 The sensor must detect if anything is between the two pillars of the gate (so between the two sensors).

REQ-03-2 If the sensor detects a movement, a *blocking* signal must be sent immediately to the *Control Logic*.

REQ-03-3 If the sensor can not detect any more the movement 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*.

2.2.4 Switch Controller

REQ-04-1 The controller should have an *open* and a *close* gate action button.

REQ-04-2 One action must be completely finished to start a new action.

2.2.5 GateStateSensor

REQ-05-1 It must detect the final closed/opened state of the *gate*, and send information about it to the *Control Logic*.

REQ-05-2 In the final states the gate can not be moved forward.

2.2.6 GarageGateLogic

REQ-06-1 The *Control Logic* can get actions from the *Switch Controller*. One action must be finished to receive new actions.

REQ-06-2 While the gates are moving (by an action from the *Switch Controller*), the *Control Logic* should stop the movement, if it gets a *blocking* sign from the *Motion Sensor* and stop the *Motor*. Consequently the *Control Logic* should continue the movement, if it gets a *free* sign from the *Motion Sensor* and start the *Motor*.

REQ-06-3 The *Control Logic* can receive the signals from the *Motion Sensor*, *Lamp*, *Switch Controller* and *Gate State Sensor*.

REQ-06-4 The *Control Logic* must communicate to each component with dedicated signal types.

2.3 System model representation

Communication between blocks In this section I want demonstrate the logical behaviour and communication constraints for the Garage Gate sample system. The component communication is described with the following figure 2.2.

The sensors and subsystems connects to the *GarageGateLogic*, to its dedicated ports. Basically the *SwitchController* ,*MotionSensor* and *GateStateSensor* sends control signals to the *GarageGateLogic*, besides the *Lamp* and *Motor* components receives control signals from the *GarageGateLogic*.

Logical operation 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. First of all we can start from the initial *Closed* state, where

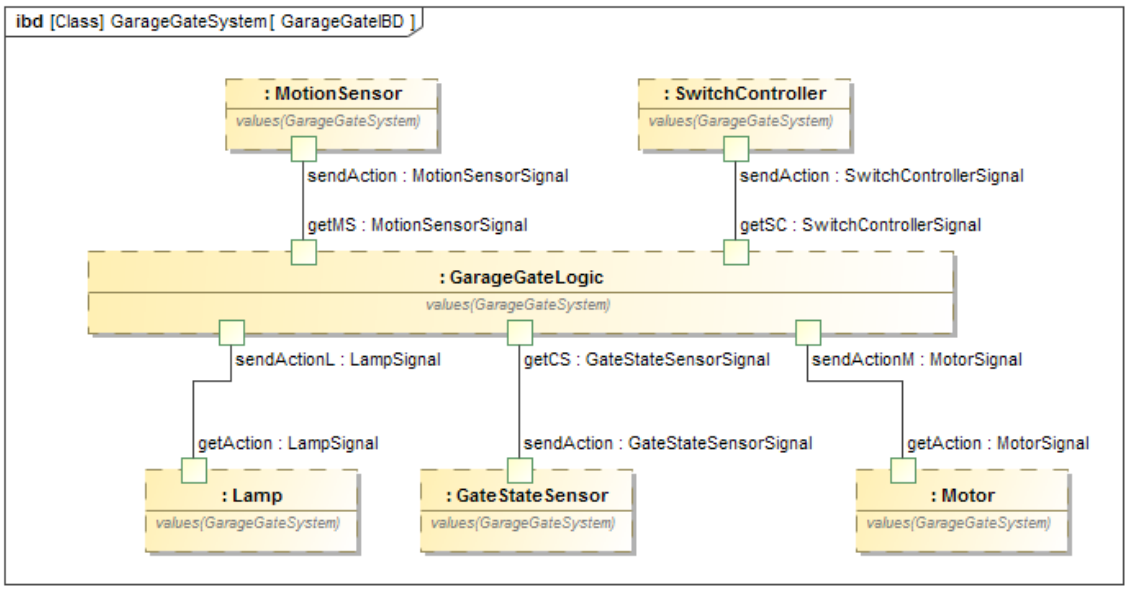

Figure 2.2. Gate controller internal block diagram

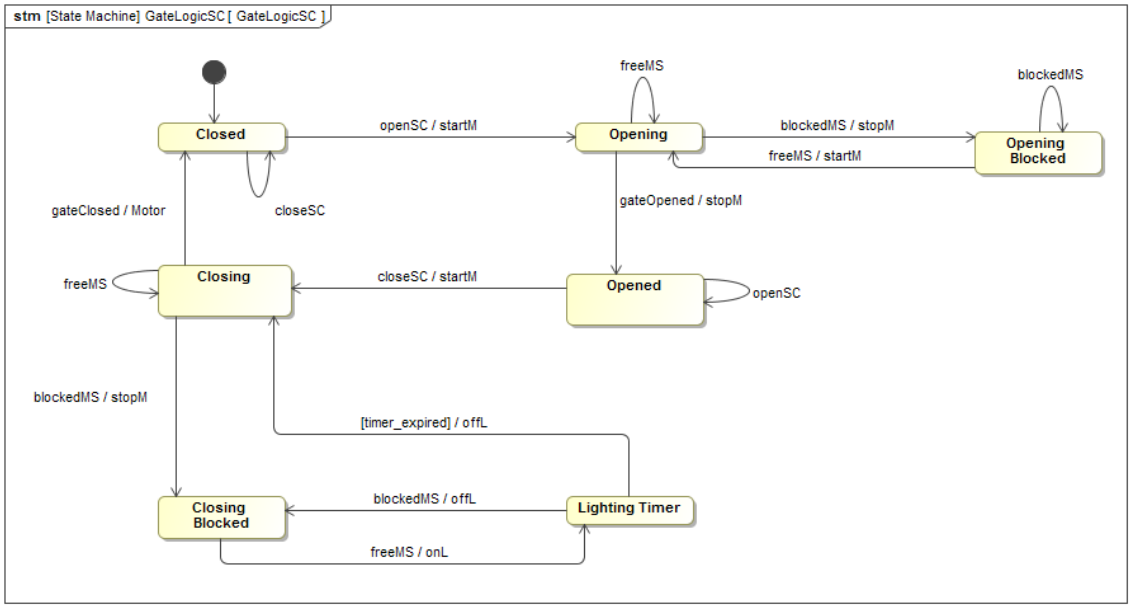

Figure 2.3. Garage gate state machine diagram

the *Switch Controller* can open the gate with an 'open' command. This command sets the state machine in an *Opening* state, while starting the motor functions. While opening the gate, somebody or something can move into the way of the gate, so this becomes *Block Opening*. The gate is opening again, if the blocking object moved out of the way consequently it is free. After the *Opening* phase succeeded the gate is *Opened*.

In this state we can 'close' the gate with a simple command by the *Switch Controller*, 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* and the motor has been stopped.

Testing goals I have decided to test the proper functionality of the whole garage gate system. Therefore the test scenario is to achieve all states in the state-machine model and to test most of the transitions from that.

Chapter 3

Testing testing tools

3.1 Garage Gate System Test Implementation

3.1.1 System implementation

In order to discover different testing tools and environments, I have implemented the Garage Gate sample in c# with Visual Studio and in java with Eclipse. In the 2.3 section described the sample's state machine, which is quite straight-forward to implement, consequently I will focus in this section the known bugs in the code, which I want to detect by testing. For testing goals by the implementation I have omitted some transitions from the state machine (see 2.3. figure), which were the loops (a transition which source and target state is the same), like in Closed state the closeSC and in Opening state the FreeMS.

3.1.2 Test implementation

GraphWalker test structure We can easily set up the GraphWalker test structure with Maven for our existing project. Then GraphWalker generate test cases from a graphml file, which is added to the project, thus I have created a graph representation matching the full functionalities of the SUT (see 3.1. figure)

With the *graphwalker:generate-sources* command the tool have created an adaptation layer for our SUT. For testing purposes I configured *GraphWalker* with the following annotation: @GraphWalker(start = "e_init", value = "quick_random(edge_coverage(80))"). With this command the test starts with the transition named "e_init" and stops if the edge coverage value is above 80%. The tool supports several algorithm like quick_random, random or a_star.

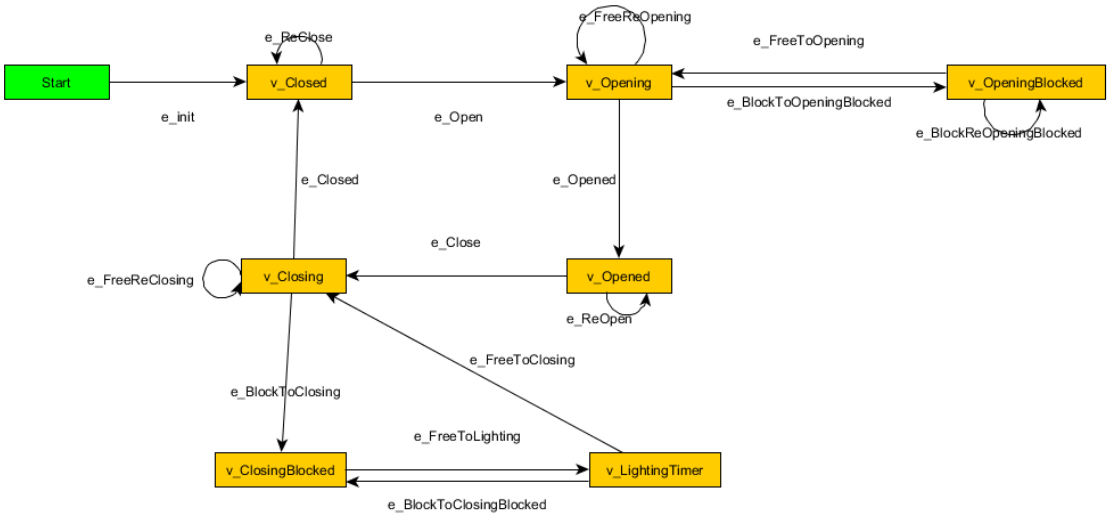

Figure 3.1. GraphWalker graph test model

In the adaptation code the recommended scenario is to have all commands to the SUT through the edge, and we assert our state in the vertexes. Sample code snippet:

```
@GraphWalker(start = "e_init", value = "quick_random(edge_coverage(80))")
public class GateModelTest extends ExecutionContext implements GateModel {
private GarageGateLogic gate;

@Override
public void e_BlockToOpeningBlocked() {
gate.setGateState(GarageGateState.OPENING);
gate.block();

}

@Override
public void v_Opening() {
Assert.expect(gate.getGateState()).equals(GarageGateState.OPENING);
}
```

SpecExplorer test structure SpecExplorer structure provides us 3 visual studio projects for testing purposes.

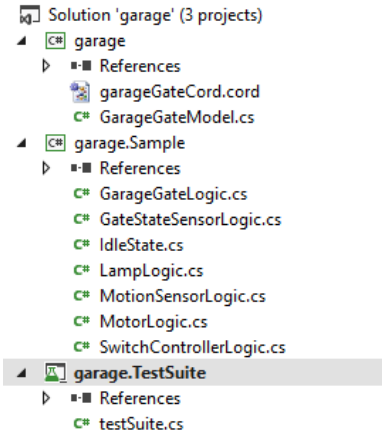

Figure 3.2. SpecExplorer Visual Studio testing structure

- **garage project:** Contains the configuration of SpecExplorer tool with cord files and belonging models.
- **garage.Sample:** This project can contain the whole implementation code (like in our case) or an test adapter which connects to the real SUT.
- **garage.TestSuite:** Unit test project. It is a result by test generation process.

To create a test suite which is satisfies our purposes, we need to create a finite cord file. First we list the available actions in a configuration:

```
config Config
{
        action abstract static void GarageGateLogic.openGate();
        action abstract static void GarageGateLogic.closeGate();
        action abstract static void GarageGateLogic.gateClosed();
        action abstract static void GarageGateLogic.gateOpened();
        action abstract static void GarageGateLogic.free();
        action abstract static void GarageGateLogic.block();

        switch StepBound = 128;
        switch PathDepthBound = 128;
        switch TestClassBase = "vs";
        switch GeneratedTestPath = "..\\garage.TestSuite";
        switch GeneratedTestNamespace = "garage.TestSuite";
        switch TestEnabled = false;
        switch ForExploration = false;
}
```

The next step is to create test-runnable machines for our needs, thus the state space get traversal by the following machine step by step:

```
machine gateSteps() : Config where ForExploration = true
{
(openGate; free; block; free; gateOpened; closeGate; free; block; free; free; gateClosed;)*
}
machine testSuite() : Config where ForExploration = true, TestEnabled = true
{
construct test cases where Strategy = "ShortTests" for gateSteps()
}
```

Our model file is calling the implementation from the garage project. Each action in the cord file can contract with a method in the model program. A sample code shown here:

```
class GarageGateModel
{
private GarageGateState state;

[Rule(Action = "openGate()")]
public void openGate()
{
GarageGateLogic.openGate();
        state = GarageGateLogic.GateState;
}
```

We can now explorer and run our testSuite machine from the Exploration Manager. (see in 3.3. figure)

3.2 Testing results

All the testing tools found the bugs in the implementation, which was consciously written by me. The *SpecExplorer* gives direct navigation to the source code, where the exception was thrown, (see 3.4.).

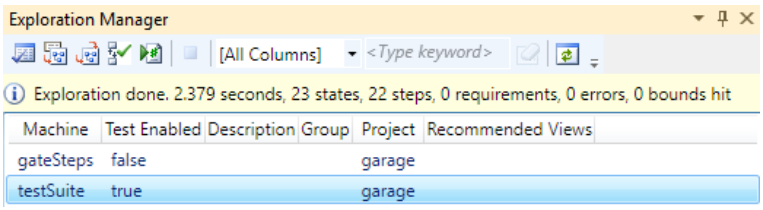

Figure 3.3. Test environment in Visual Studio

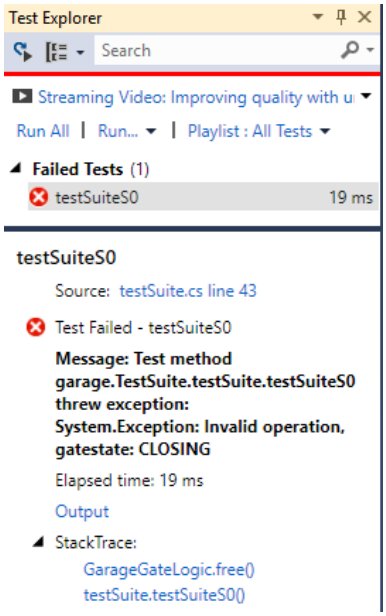

Figure 3.4. Test execution in Visual Studio

The same error was found by *GraphWalker* with random(edge_coverage(80)) settings, and we can get detailed information about the failure:

```
failures": [{"failure": "java.lang.RuntimeException\r\n\tat hu.bme.mit.GarageGate.GarageGateLogic.
    free(GarageGateLogic.java:78)\r\n\tat hu.bme.mit.GarageGate.GateModelTest.e_FreeReClosing(
    GateModelTest.java:88)\r\n\tat jdk.nashorn.internal.scripts.Script$Recompilation$22$369$\\^eval
    \\_.e_FreeReClosing(<eval>:1)\r\n\tat jdk.nashorn.internal.scripts.Script$21$\\^eval\\_.:program
    (<eval>:1)\r\n\tat jdk.nashorn.internal.runtime.ScriptFunctionData.invoke(ScriptFunctionData.
    java:637)\r\n\tat jdk.nashorn.internal.runtime.ScriptFunction.invoke(ScriptFunction.java:494)\r\
    n\tat jdk.nashorn.internal.runtime.ScriptRuntime.apply(ScriptRuntime.java:393)

"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 0,
"totalNumberOfVisitedEdges": 5,
"totalIncompleteNumberOfModels": 0,
"vertexCoverage": 57,
"totalNumberOfEdges": 16,
"totalNumberOfVisitedVertices": 4,
"edgeCoverage": 31,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 11
```

Interesting fact that with quick_random(edge_coverage(80)) setting the same bug was not found, with the same adaptation code and implementation, just with higher edge coverage percentage. The Test result is:

```
[INFO] Result :
[INFO]
[INFO] {
"totalFailedNumberOfModels": 0,
"totalNotExecutedNumberOfModels": 0,
```

```
"totalNumberOfUnvisitedVertices": 0,
"verticesNotVisited": [],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 13,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [
{
"modelName": "GateModel",
"edgeId": "e2",
"edgeName": "e_FreeReClosing"
},
{
"modelName": "GateModel",
"edgeId": "e13",
"edgeName": "e_BlockReOpeningBlocked"
},
{
"modelName": "GateModel",
"edgeId": "e14",
"edgeName": "e_ReClose"
}
],
"vertexCoverage": 100,
"totalNumberOfEdges": 16,
"totalNumberOfVisitedVertices": 7,
"edgeCoverage": 81,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 3
}
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 18.013 s
[INFO] Finished at: 2017-05-22T22:44:40+02:00
[INFO] Final Memory: 46M/446M
```

Reaching test goal In *Spec Explorer* after detecting all the bugs in the implementation code, the test results in a passed state. (see 3.5.).

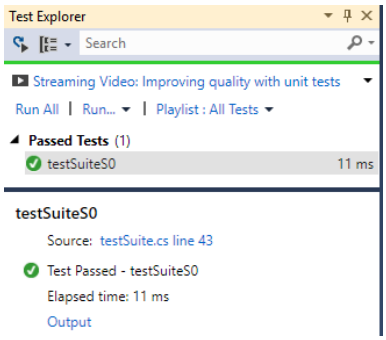

Figure 3.5. Test execution in Visual Studio

In *GraphWalker* with setup quick_random(vertex_coverage(100)) test settings the result was successful, although with random(vertex_coverage(100)) the result was failure, because there were still missing loops in the implementation so there could be further development and investigating processes.

```
quick\_random(vertex\_coverage(100))
[INFO] Result :
[INFO]
[INFO] {
"totalFailedNumberOfModels": 0,
```

```
"totalNotExecutedNumberOfModels": 0,
"totalNumberOfUnvisitedVertices": 0,
"verticesNotVisited": [],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 8,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [
{
"modelName": "GateModel",
"edgeId": "e1",
"edgeName": "e_Closed"
},
{
"modelName": "GateModel",
"edgeId": "e2",
"edgeName": "e_FreeReClosing"
},
{
"modelName": "GateModel",
"edgeId": "e6",
"edgeName": "e_BlockToClosingBlocked"
},
{
"modelName": "GateModel",
"edgeId": "e7",
"edgeName": "e_FreeToClosing"
},
{
"modelName": "GateModel",
"edgeId": "e10",
"edgeName": "e_FreeReOpening"
},
{
"modelName": "GateModel",
"edgeId": "e13",
"edgeName": "e_BlockReOpeningBlocked"
},
{
"modelName": "GateModel",
"edgeId": "e14",
"edgeName": "e_ReClose"
},
{
"modelName": "GateModel",
"edgeId": "e15",
"edgeName": "e_ReOpen"
}
],
"vertexCoverage": 100,
"totalNumberOfEdges": 16,
"totalNumberOfVisitedVertices": 7,
"edgeCoverage": 50,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 8

**************************************************************************
random(vertex\_coverage(100))

[INFO] Result :
[INFO]
[INFO] {
"totalFailedNumberOfModels": 0,
"totalNotExecutedNumberOfModels": 0,
"totalNumberOfUnvisitedVertices": 0,
"verticesNotVisited": [],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 14,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [
{
```

```
"modelName": "GateModel",
"edgeId": "e12",
"edgeName": "e_FreeToOpening"
},
{
"modelName": "GateModel",
"edgeId": "e13",
"edgeName": "e_BlockReOpeningBlocked"
}
],
"vertexCoverage": 100,
"totalNumberOfEdges": 16,
"totalNumberOfVisitedVertices": 7,
"edgeCoverage": 87,
"totalNumberOfVertices": 7,
"totalNumberOfUnvisitedEdges": 2
}
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 4

Summary

Through the Project Laboratory 2 subject I have tried out two different testing environments and learned a lot about the test execution strategies. I have created a common example project with well-documented specification.

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`.