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

Test generation

MSc Project Laboratory 1

Author
Gergő Ecsedi

Advisor
dr. Zoltán Micskei

December 4, 2016

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Garage Gate **2**

2.1 State machine introduction 2

3 GraphWalker **5**

3.1 GraphWalker 5

3.2 GraphWalker implementation in Eclipse 5

List of Figures **7**

List of Tables **8**

Appendix **9**

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

Garage Gate

2.1 State machine introduction

The system consists 4 elements, which are shown on the 2.1. 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.

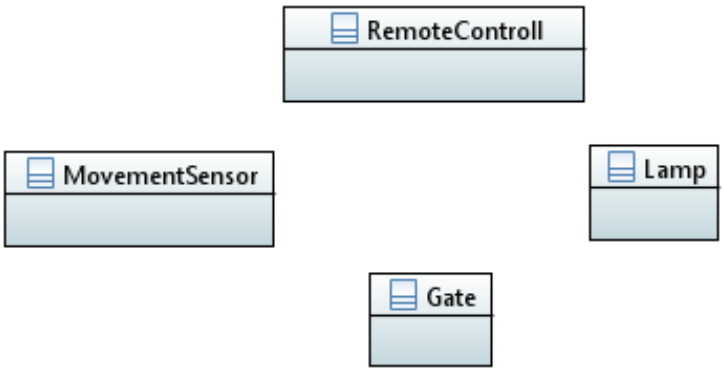

Figure 2.1: Garage gate components

These components can communicate to each other directly. The possible communication messages is shown on the 2.2.

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

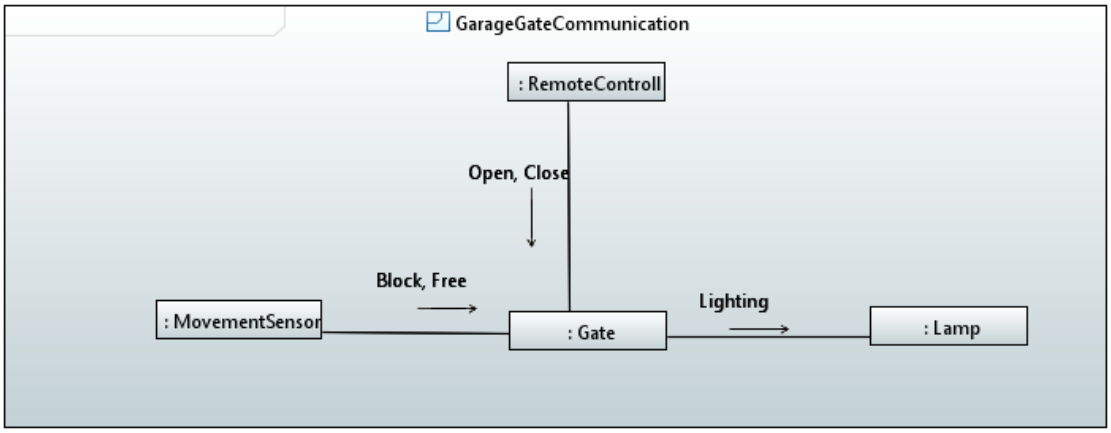

Figure 2.2: Garage gate communication diagram

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*.

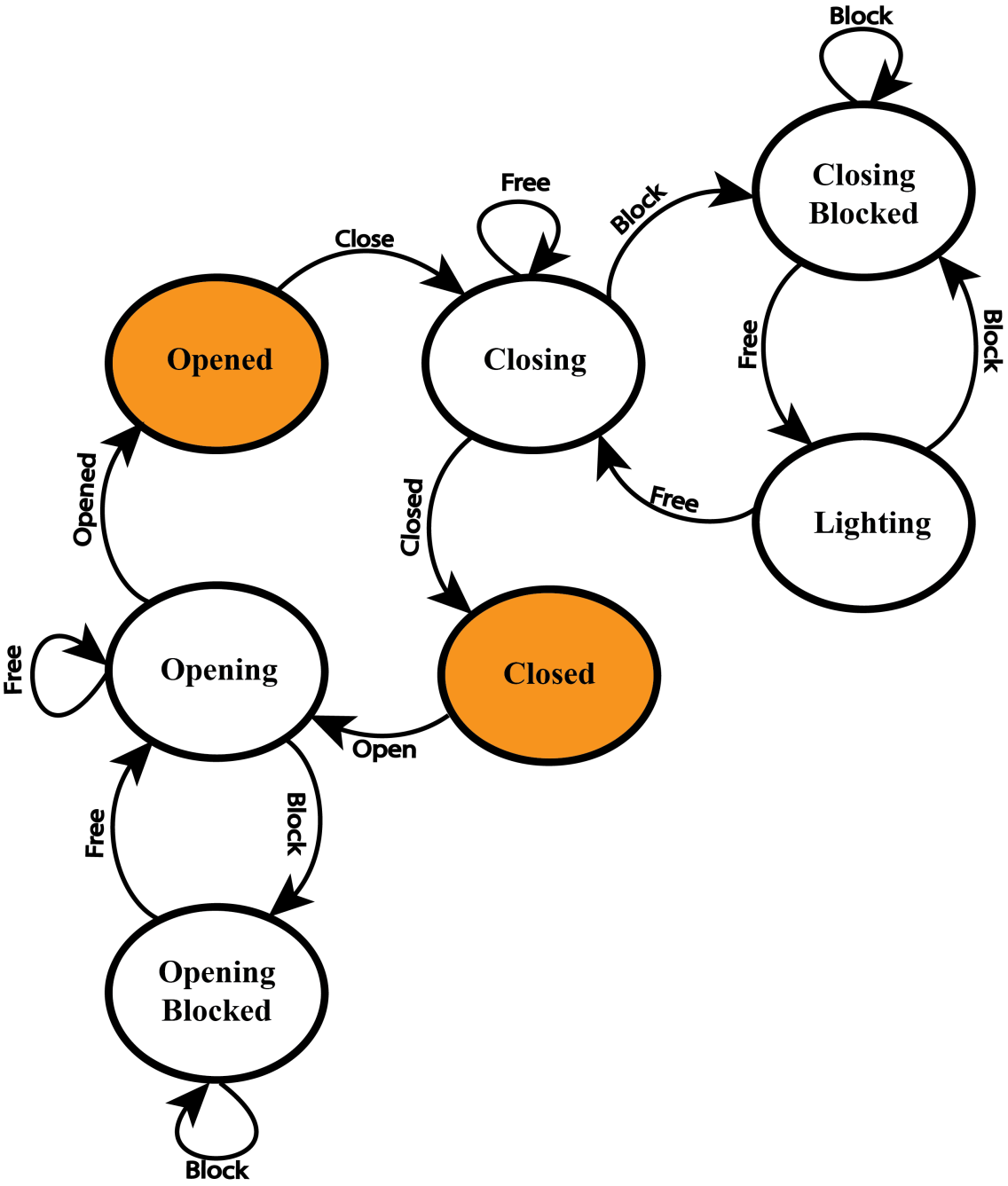

Figure 2.3: Garage gate state machine diagram

Chapter 3

GraphWalker

3.1 GraphWalker

GraphWalker is a open source Model-based testing tool for test automation. It's designed to make it easy to design 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). A path is used to call the corresponding methods or functions of your SUT (system under test). 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.

There are two ways to execute 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.

3.2 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 2.3.), and a graph model for the lamp, which called from the previous graph.

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, "totalIncompleteNumberOfModels": 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

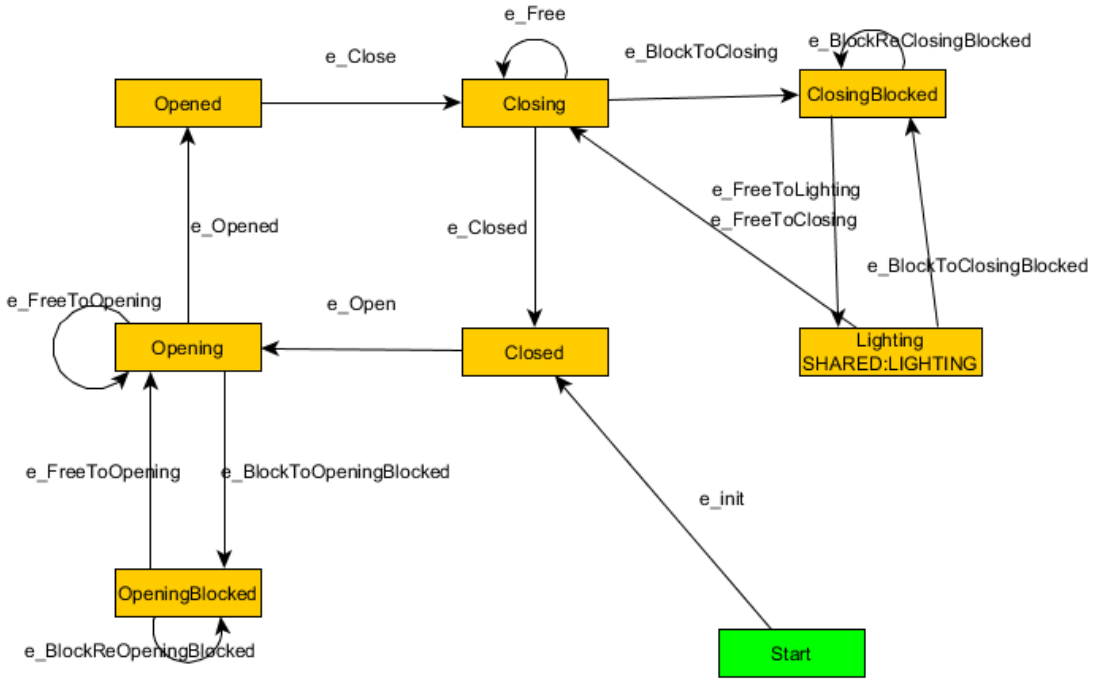

Figure 3.1: Garage gate model with yEd

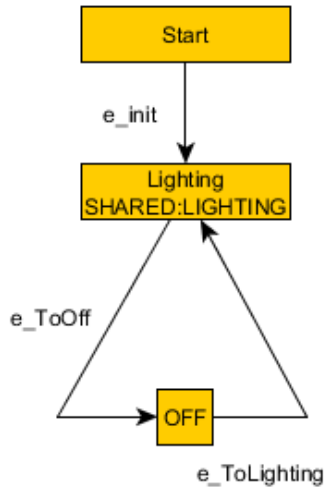

Figure 3.2: Lamp0 model with yEd

List of Figures

2.1 Garage gate components 2
2.2 Garage gate communication diagram 3
2.3 Garage gate state machine diagram 4

3.1 Garage gate model with yEd 6
3.2 Lamp0 model with yEd 6

List of Tables

Appendix