Final Thesis

Automated Software Testing in an Embedded Real-Time System

by

Johan Andersson
and
Katrin Andersson

LITH-IDA-EX--07/046--SE

2007-08-17

Linköping University
Department of Computer and Information Science

Final Thesis

Automated Software Testing in an Embedded Real-Time System

by

Johan Andersson
and
Katrin Andersson

LITH-IDA-EX--07/046--SE

2007-08-17

Supervisor: Mariam Kamkar, Linköping University
Guido Reinartzt, IVU Traffic Technologies AG
Examiner: Mariam Kamkar, Linköping University

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2007-08-16

Linköpings universitet

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN

ISRN LITH-IDA-EX--07/046--SE

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

http://www.ep.liu.se/

Titel
Title

Automated Software Testing in an Embedded Real-Time System

Författare
Author

Johan Andersson
Katrin Andersson

Sammanfattning
Abstract

Today, automated software testing has been implemented successfully in many systems, however there does still exist relatively unexplored areas as how automated testing can be implemented in a real-time embedded system. This problem has been the foundation for the work in this master thesis, to investigate the possibility to implement an automated software testing process for the testing of an embedded real-time system at IVU Traffic Technologies AG in Aachen, Germany. The system that has been the test object is the on board system i.box.

This report contains the result of a literature study in order to present the foundation behind the solution to the problem of the thesis. Questions answered in the study are: when to automate, how to automate and which traps should one avoid when implementing an automated software testing process in an embedded system.

The process of automating the manual process has contained steps as constructing test cases for automated testing, analysing whether an existing tool should be used or a unique test system needs to be developed. The analysis, based on the requirements on the test system, the literature study and an investigation of available test tools, lead to the development of a new test tool. Due to limited devlopement time and characterstics of the i.box, the new tool was built based on post execution evaluation. The tool was therefore divided into two parts, a part that executed the test and a part that evaluated the result.. By implementing an automated test tool it has been proved that it is possible to automate the test process at system test level in the i.box.

Nyckelord
Keywords

Automated software testing, embedded systems, software test procedure, software testing, on board integrated system.

ABSTRACT

Today, automated software testing has been implemented successfully in many systems, however there does still exist relatively unexplored areas as how automated testing can be implemented in a real-time embedded system. This problem has been the foundation for the work in this master thesis, to investigate the possibility to implement an automated software testing process for the testing of an embedded real-time system at IVU Traffic Technologies AG in Aachen, Germany. The system that has been the test object is the on board system i.box.

This report contains the result of a literature study in order to present the foundation behind the solution to the problem of the thesis. Questions answered in the study are: when to automate, how to automate and which traps should one avoid when implementing an automated software testing process in an embedded system.

The process of automating the manual process has contained steps as constructing test cases for automated testing, analysing whether an existing tool should be used or a unique test system needs to be developed. The analysis, based on the requirements on the test system, the literature study and an investigation of available test tools, lead to the development of a new test tool. Due to limited devlopement time and characterstics of the i.box, the new tool was built based on post execution evaluation. The tool was therefore divided into two parts, a part that executed the test and a part that evaluated the result.. By implementing an automated test tool it has been proved that it is possible to automate the test process at system test level in the i.box.

ACKNOWLEDGEMENTS

Many people have helped us making this report to what is has become. It has been a real pleasure to have gotten the splendid opportunity to carry out our master thesis at IVU Traffic Technologies AG in Aachen, Germany, and at the same time been able to put ourselves right into the adventure it was coming like aliens to another country. We particularly want to direct our thanks to Dik Lokhorst for having the confidence in us carrying out this interesting project, to Guido Reinartz and Dieter Becker for guiding us through the maze of implementing a new testing process, to Wolfgang Carius not only for always helping us when we bothered him with our troublesome requests, but also for always doing so with a smile, to Peter Börger for always being able to spare some time and his willingness of using his impressive programming skills to help us solve problems, to Andrea Heistermann for her interest in our work, to Oliver Lamm for introducing us to the world of public transport software, to Andreas Küpper for friendly conversations that made us feel like home. We would also like to thank our college Matthieu Lux for pleasant collaboration and for many exciting discussions during the lunches, our examiner Mariam Kamkar for giving us support and guidance, and our opponents Martin Pedersen and Johan Millving.
Please accept our apologies if we have not included anyone in this acknowledgement that we should have.

Hurghada, Egypt, July 2007

“Science and art belong to the whole world, and before them vanish the barriers of nationality.”

Goethe

TABLE OF CONTENTS

1 Introduction 1
1.1 Background 1
1.2 Problem description 2
1.3 Purpose. 2
1.4 Goal 2
1.5 Method 2
1.6 Delimitations 3

PART I: Introduction to software testing 5

2 The test procedure 7
2.1 Testing as a process 7
2.2 The V-model 8

3 Higher order testing 11
3.1 System test 11
3.1.1 Performance testing 11
3.1.2 Volume testing 12
3.1.3 Stress testing 12
3.1.4 Configuration testing 13
3.1.5 Recovery testing 13
3.2 Function testing 14
3.3 Regression testing 15

4 Techniques for creating test cases 17
4.1 Equivalence class partitioning 17
4.2 Boundary value analysis 19
4.3 Domain analysis testing 19
4.4 Decision table testing. 19

4.5 Pairwise testing 21
4.5.1 Orthogonal arrays 22

PART II: Automated software testing 27

5 Automated testing 29
5.1 Introduction 29
5.2 Benefits of automated testing 30
5.3 Drawbacks of automated testing 30

6 Automating a manual test procedure 33
6.1 Deciding when to automate 33
6.2 Creating test cases for automated testing. 35
6.3 Test performance 35
6.3.1 Scripting techniques 35
6.4 Test evaluation. 38
6.4.1 Simple and complex comparison 38
6.4.2 Sensitivity of test 38
6.4.3 Dynamic comparison 38
6.4.4 Post-execution comparison 39
6.5 Test result. 39

7 Automated testing in embedded systems. 41
7.1 Definition of an embedded system 41
7.2 Embedded software vs. regular software. 42
7.3 Defining the interfaces. 43
7.4 Signal simulation 43
7.4.1 Full simulation 43
7.4.2 Switched simulation 44

PART III: Implementing automated testing . . 45

8 Description of the system 47
8.1 IBIS 47
8.2 The i.box 47
8.2.1 Positioning 48
8.2.2 Characteristics suitable for test automation . . . 51

8.2.3 Characteristics not suitable for test automation 52

9 Automating the test process. 53
9.1 Test method 53
9.1.1 Full signal simulation vs. switched signal simulation 54
9.1.2 Scripting technique 54
9.1.3 Dynamic comparison vs. post execution comparison 54
9.1.4 Simple and complex comparison 55
9.2 Constructing test cases for automated testing 55
9.2.1 Administrative information 58
9.2.2 Preconditions 58
9.2.3 Actions 59
9.2.4 Expected output 63
9.3 Analysis of existing test tools 65
9.3.1 Requirement on tool 65
9.3.2 Mercury Functional Testing for Wireless 66
9.3.3 Automation Anywhere 67
9.3.4 Other tools 67
9.3.5 Result of analysis 68
9.4 Design of test execution. 68
9.4.1 Auster 68
9.4.2 B4 test system 70
9.4.3 Design overview 71
9.4.4 Design patterns 76
9.4.5 Concepts 76
9.4.6 Implementation notes to the test execution . . . 81
9.4.7 Distance calculations 82
9.5 Design test evaluation 85
9.5.1 Evaluation 86
9.6 Generate result 89
9.6.1 Format 89
9.6.2 XML file 89
9.6.3 XSL presentation 90

PART IV: Analysis and future work 93

10 Analysis 95
10.1 Result 95

11 Future work 97
11.1 Measure the automated testing 97
11.2 Dynamic comparison 98
11.3 Subscripted tests 98
11.4 Test evaluation 99
11.5 Test of another real-time embedded system 99
11.6 Extending test focus 99

Appendix A Design diagrams 101
A.1 The test performance 101
A.1.1 Internal module collaboration 101
A.1.2 Detailed class diagrams 108
A.1.3 Activity diagrams 141
A.2 The test evaluation 184
A.2.4 Detailed class diagrams 184

Bibliography 189

Glossary 191

CHAPTER 1

INTRODUCTION

This first chapter presents the background of the master thesis and important parts as the problem description, goal and aim of the work. It also contains a description of the method that has been used and the delimitations in the work.

1.1 Background

At the beginning of the era of manufacturing clothing it was truly seen as a craft. Over years it has developed from the tailor sewing by hand to large industries where machine play a major part in making the process more efficient by automating wherever possible. This evolution also reflects in industries like the software industry where making the software development process as efficient as possible by using the power of automated software testing where suitable.

This master thesis presents how the power of automated software testing can be used. When one is finding out about how automation can be used in a quality assurance process, an interesting issue that may arise is how to get there, how should one go from manual testing to automated testing? By reading this master thesis one will get information about all of the steps that needs to be passed in the process of going from manual testing to automated testing. A practical example of how it can be possible to switch from manual testing to automated testing is described. The system that is the test object is an embedded real-time system.

1.2 Problem description

This thesis will investigate the possibility to implement an automated software testing process for the testing of an embedded real-time system at IVU Traffic Technologies AG in Aachen, Germany.

1.3 Purpose

The reason for wanting to automate a part of the testing process is a desire to improve efficiency and secure a high quality of the product. The improved efficiency can be achieved by running tests at night on hardware that is not used, and by the time the test responsible spends on managing automated testing being less than the time spent on running tests manually. A higher quality is hoped to be reached through more structured and thorough testing.

1.4 Goal

This master thesis has resulted in a study of how to automate a test procedure and an implementation of an automated testing process.

1.5 Method

The main idea behind the method has been to build a knowledge base by analysing previous work within the area of automated testing to have a stable foundation when implementing an automated test procedure. To be able to achieve the goal of this master thesis the work has been structured in the following steps:

- literature studies within the area software testing and automated software testing.
- a research upon how manual system testing is done;
- formulating test cases that should be automated;
- research in existing test tools that could be useful;
- designing and implementing an automated test process;

1.6 Delimitations

The process that is automated in the master thesis is a regression testing process that is performed to verify correct functionality in parts that have not been edited and to test different combination of software and hardware. The automated testing process in the study is performed at system testing level, and it should only include testing positioning functionalities in the i.box product.

PART I

INTRODUCTION TO SOFTWARE TESTING

When going through the area of software testing it may seem as a jungle of different thoughts and declarations of concepts, the aim of this introduction part is to guide the reader through that jungle. It presents ideas that are good to have knowledge in when going from manual testing to automated testing.

The well established V-model will be presented in this part to give a picture over the whole testing procedure in product development. It will continue by focusing on higher ordered testing, since the problem to solve in this master thesis is at that level. The concept of regression testing is presented while it is an important part in the study. The part ends with some techniques on how to create test cases since creating test cases often is a part of automating a manual test process.

CHAPTER 2

THE TEST PROCEDURE

To succeed with the development of a software, the developers need to follow some structured procedure. This also holds for the testing of the software, hence a test procedure is crucial for the success of the testing.

2.1 Testing as a process

The software development process is described as a series of steps with the aim of reaching a finished software product. Embedded within the software development process are several other processes where the testing process is one of them. Testing also includes two other processes called validation and verification.

"Validation is the process of evaluating a software system during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements." [IEEE 1990]

"Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase." [IEEE 1990]

Validation is usually done by exercising the code with test cases and verification is usually done by inspections and reviews of software deliverables. The testing process covers both validation and verification and includes all of the following: technical reviews, test planning, test tracking, test case design, unit test, integration test, system test, acceptance test and usability test [Burnstein 2003]. This thesis covers the subject of test case design and validation at system test level.

2.2 The V-model

To support the engineers during the development of software, some model is usually used. One of the more commonly used is the V-model which is an extended variant of the more basic waterfall model. ”Figure 2.1: The V-model [Burnstein 2003].” on page 9 shows a picture of the testing process integrated with the software development process as it is exercised using the V-model.

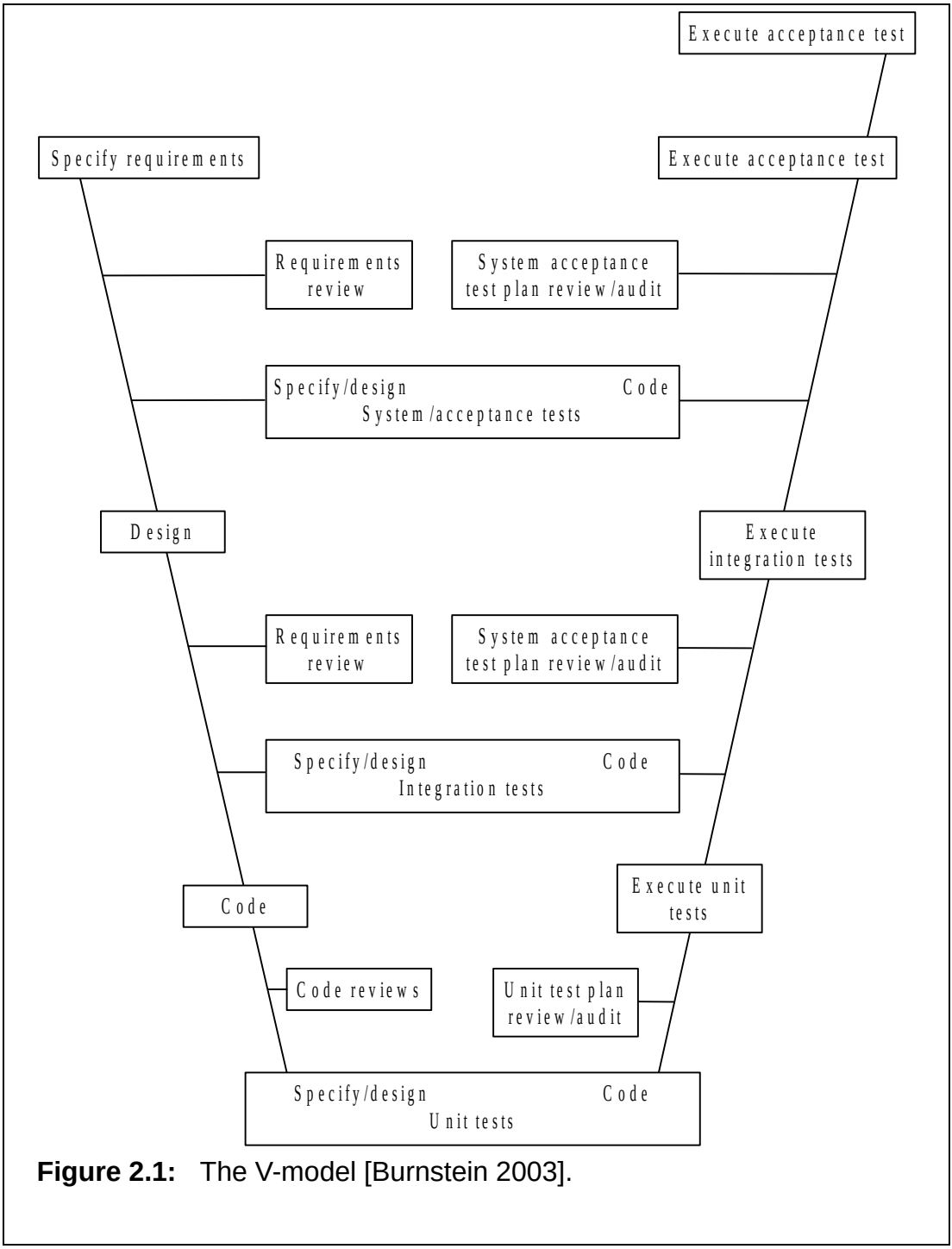


Figure 2.1: The V-model [Burnstein 2003].

CHAPTER 3

HIGHER ORDER TESTING

When the units that build up the software are tested seperately during the unit testing, it is time to put the units together and test their collaboration with each other. This testing is called integration testing and is a testing procedure on a lower level than the testing this thesis deals with, namely higher order testing.

3.1 System test

After every subsystem has been developed and its functions tested individually, the subsystems are put together into the final system and tested as a group. This testing is referred to as system testing and exists as several types of testing. Not all software systems need to undergo all these types of system tests. It is up to the test designers of the system testing phase to decide which test types are applicable for the specific software [Burnstein 2003]. For example, if multiple device configurations is not a requirement for the system, then the need for configuration testing is not significant.

3.1.1 Performance testing

In order to be sure that the developed system is really usable, one have to assure that the system for example responds in a certain amount of time, handles a specified number of transactions per minute or uses no more memory than specified. Otherwise, the end user might find the system unusable. For example, a user wants to write a new e-mail in a web based e-mail client so she clicks on the compose new message button and expects the window representing

the new message to be displayed. But for some reason, every second attempt to create a new message causes the system to have a response time of over one minute. In that case, the user would most certain search for another e-mail client. The purpose of performance testing is to find such defects in the system according to specified requirements. It is very important that those requirements are measurable and not only vaguely specified as: “The system should respond in a reasonable amount of time”. In such a case the test would have an outcome that very much depends on the person running the test case.

3.1.2 Volume testing

Another type of system testing is loading the software with extremely large data. A word processor could for example be loaded with a document containing several thousands of pages, and an automated test tool could be loaded with a test case with absurdly many actions. The purposes of volume testing are to show that the software cannot handle the volume of data specified in its requirements and to find how much data it can handle. [Myers 2004]

3.1.3 Stress testing

When a system is input with a large load of data in a short period of time, it is called a stress test. The purpose is trying to find in which situation the system breaks. Stress testing should not be confused with volume testing. “A heavy stress is a peak volume of data, or activity, encountered over a short span of time” [Myers 2004]. For example, when stress testing a telephone switching software, one could simulate 200 incoming new calls under the period of one second. Stress tests are particularly significant when testing real-time systems and systems handling various degrees of load. Stressing the control software in an air plane could involve giving full throttle, pulling the nose up, raising the landing gear, banking right, lowering the flaps and deactivating the autopilot, all at the same time. This looks like a situation that will never occur, because of the physical impossibility of the human pilot to carry out all these tasks with only two hands and two feet. There still exist a value in testing such a situation because if the test would detect an error in this “will never oc-

cur" situation, it is also likely that the same deficiency also will show in more realistic, less stressful situations [Myers 2004].

3.1.4 Configuration testing

Often, systems are required to run under different types of software or hardware configurations and they have to be tested on all possible configurations in order to be sure that they fulfil the configuration requirements. When developing a web based application for example, the system can have requirements that it can be run in one or many web browsers. If specified to run in different browsers, the application has to be tested in all of these browsers since they differ in the way they are implemented. In addition, the same web browser can operate differently depending on under which operation system it is run. When writing test cases for configuration testing, tests should be focused to find defects in the areas where the different platforms are known to have differences.

Users often require that devices that the software interacts with, such as printers, must be interchangeable, removable or reconfigurable. Often, the software has some menu or set of command that allows the user to specify which kind of device is in use. According to Beizer several types of operation should be performed to test that the software meets its configuration requirements [Beizer 1990]: change the positions of devices, include errors in each of the device and include errors in several devices to see how the system reacts.

3.1.5 Recovery testing

The design purpose of systems with recovery facilities is to minimize the mean time to recovery, because downtime often causes the company to loose money since the system is inoperable [Myers 2004]. When performing recovery testing the system is subjected to device failures in order to check if the software can detect such device failures and continue from a known state. An ATM, for example, has to be able to handle the loss of connection to the bank during a money withdrawal request. Otherwise, the user might find herself in a situation where the transaction was sent from the ATM and registered in the bank's server, but the confirmation was never received by the

ATM. So the user never gets her money and still has a registered withdrawal in her account.

Some systems rely on different input sources to calculate their output. They can be designed to calculate a result of varied precision depending on how many input sources are functioning. When testing such a system's ability to recover, one can simulate loss of input on some sources and check if the system detects the failing input, but still produces a correct output.

3.2 Function testing

Function testing is sometimes considered a part of system testing, for example in the point of view of Burnstein [Burnstein, 2003]. Others, for example Myers, look upon function testing as a test phase separated from the system testing phase [Myers, 2004].

The function testing is performed in an attempt to find discrepancies with the specific description of the functionality of the system according to the perspective of the end user. The objective is not to prove that the software is correct, it is to find errors [Myers, 2004]. Otherwise, function testing would only be running the acceptance test at an earlier moment in time. Instead, function testing is performed so that no error remains in the system at the acceptance testing phase and discovered with the customer leaning over your shoulder in that late stage of the development.

During the function testing phase, new test cases would have to be created, but it is also of great benefit to reuse test cases run during earlier phases, for example during unit and integration testing, [Watkins 2001]. Function testing is completely carried out as black-box testing and several useful techniques exist to help creating test cases used in function testing. These techniques include: equivalence-class partitioning, boundary value analysis, decision table testing, pair wise testing, state-transition testing, domain analysis testing, and use case testing. Those that are applicable for this thesis are further described in “4 Techniques for creating test cases” on page 17.

3.3 Regression testing

Regression testing is not defined as a level of testing, but it is the re-testing of software that occurs when changes are made to ensure that the software still works according to the specification. Regression testing can be implemented at any level of test. According to Burnstein, regression testing is especially important when multiple software releases are developed [Burnstein 2003]. The users want new features but they still want the old functions to be working as earlier specified. The only purpose of regression tests is to determine if new code has corrupted, or “regressed” old functions [Loveland 2004].

CHAPTER 4

TECHNIQUES FOR CREATING TEST CASES

One important matter in testing is: how to find which input values to test with in the test cases? There exist a number of techniques to use when testing with black-box testing and white-box testing. This thesis concentrates on function testing on system level and therefore presents theory for how to create test cases for black-box testing.

4.1 Equivalence class partitioning

To reduce the number of test cases to run, equivalence class testing can be used. For example, if the same output is expected for all input to a text box where the value is less than 12 and the same output is expected for all values greater than 13, then there exist two equivalence classes. Because all values in an equivalence class is said to have the same output, only one value from each equivalence class will have to be selected to be used as input in the test [Copeland 2004]. Equivalence class testing can only be used if it is known that the implementation uses ranges to decide outputs and do not assign values dependent on each specific input. ”Figure 4.2: Example of when equivalence class partitioning is applicable.” on page 18 shows an example of where equivalence class is applicable. If on the other hand, the code was written like ”Figure 4.3: Example of when equivalence class partitioning is not applicable.” on page 18, each individual value would have to be tested separately, since there exist different conditions for each value. Hopefully, most programs are

written more like the former example so in that case, the number of test cases that has to be performed is reduced to only two.

```
if (x < 13)
    y = 3;
if (x > 12)
  y = 5;
```

Figure 4.2 Example of when equivalence class partitioning is applicable.

```
if (x == -39)
    y = 3;
if (x == -38)
    y = 3;
...
if (x == 12)
    y = 3;
if (x == 13)
    y = 5;
if (x == 14)
    y = 5;
if (x == 15)
    y = 5;


...
```

Figure 4.3 Example of when equivalence class partitioning is not applicable.

4.2 Boundary value analysis

Boundary value testing extends equivalence class testing to focusing the testing on where the most errors occur, which is around the borders for boundaries [Copeland 2004]. When using boundary value analysis one start with, like for equivalence class testing, identifying the equivalence classes and then finding the boundaries for the variables. The next step is to create test cases where each variable has a value just inside the boundary, a value on the boundary and a value just outside the boundary and where each combination of those values for all parameters are tested.

4.3 Domain analysis testing

Domain analysis builds on boundary value analysis and equivalence class testing. Like those techniques, the test method is used for finding wrong behaviours around the border for a rule for a specific variable. But in opposite to boundary value analysis, where the system is tested with all possible combinations around the boundary for the variables, it holds all variables' values at values inside the boundary and varies one variable at the time to be outside and on its border. One therefore ends up with two test cases per variable. This technique is useful when there are many variables in the system that need to be tested and the number of test cases when using for example boundary value analysis will be too large. [Copeland 2004]

4.4 Decision table testing

Decision tables are used to describe a system's behaviour on certain inputs in a structured way. They can also serve as a guide to creating test cases. A decision table is built up of a set of inputs, called conditions and a set of behaviours, called actions, that represent the actions that the system should take when given a certain input. Those conditions and actions build up the base for complex business rules

for a system. In general, a decision table looks like ”Table 4.1: A general decision table.” on page 20, [Copeland 2004].

Table 4.1 A general decision table.

	Rule 1	Rule 2	...	Rule p
Conditions				
Condition 1				
Condition 2				
...				
Condition n				
Actions				
Action 1				
Action 2				
...				
Action m				

To be able to use the decision table when testing a system, one will have to fill it with values according to the business rules. The first step is to put the system inputs in the condition cells, one for each input. From here you continue with putting all outputs in the “Action 1” to “Action m” cells. Once one have all the conditions and actions representing the business rules in this general form, it is time to fill the table with the actual values for the rules. For each condition, you want to combine each of its possible values with every possible value for each other condition.

To clarify the theory, below is an example of a filled decision table for a system that is used in a auto insurance company that gives discount to people that are married and good students. It also only gives insurancies to people that are either married or a good student.

The example follows from [Copeland 2004] with some modifications.

Table 4.2 An example of a completed decision table.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Married?	Yes	Yes	No	No
Good student?	Yes	No	Yes	No
Actions				
Discount ($)	60	25	50	0
Insurance?	Yes	Yes	Yes	No

The system in this example has two conditions: “married” and “good student”, which both take the inputs “yes” or “no”. The actions for this system is “discount”, which can have numerical values, and “insurance”, which can only have the values “yes” or “no”.

In the example above, all inputs only had binary values. This makes things a lot easier when creating the decision table, while it can be filled with every possible combination of input values. But what about when the inputs can take any integer as a input? Then it is certainly impossible to take every combination of inputs and put them into the decision table. In this case, one have to analyse the system’s business rules before creating the decision table.

4.5 Pairwise testing

Pairwise testing is used to reduce the number of test cases that are to be run on a system that has many input parameters that each can take a large number of values. Since it in such a case would require a very large set of test cases to test all combinations of input parameters and values, the task to run all those tests would be unbearable. One could try to run tests for all combinations, but it would probably take such long time that the whole project would be at risk. But on the other hand, one would certainly not like to run so few tests that the risk of

leaving undetected deficiencies in the code would be too high. So one approach could be to test all pairs of variables. This is where pairwise testing enters the picture. By using this approach in a case where the system has 4 input variables that each can take 3 different values, the number of test cases to run would be reduced from 3^4=81 to 9, and in a system with 13 variables with 3 possible values each, the number of test cases would be reduced from 3^{13}=1,594,323 to only 15 tests.

Some believe that most deficiencies are either single-mode deficiencies or double-mode deficiencies, i.e. either there is an error when entering a sepcific value for an input, or the combination of the value for the input with some value for another input does not work [Copeland 2004]. When using pairwise testing, all such defects would be revealed. Pairwise testing is therefore a good approach when trying to reduce the number of test cases to run but still trying to keep the number of revealed defects on a high level.

4.5.1 Orthogonal arrays

For finding test cases that test all combinations of pairs, something called orthogonal arrays can be used, which originate from Euler as Latin Squares [Copeland 2004]. An orthogonal array is a table where each column represents the input variables, the rows represent test cases and the values in the cells represents the values for each variable in the test case representatively. If all test cases are executed, it is assured that all pairwise combinations of possible values for the input parameters are tested. To find test cases using orthogonal arrays, do the following steps:

1. Identify the variables.
2. Determine the number of choices for each variable.
3. Find an orthogonal array that fits the preconditions.
4. Map the problem on the orthogonal array.
5. Construct test cases.

Consider the following example from [Copeland 2004]:

A company is developing a web based application that should be able to run in different browsers, with different plug-ins, different

client operating systems, different web servers, and different server operating systems.

Step 1 is to identify the variables: browser, plug-in, client OS, web server, and server OS, in total five different variables.

Step 2 is to determine the number of choices for each variable. The user should be able to use Internet Explorer 5.0, 5.5, and 6.0, Netscape 6.0, 6.1, and 7.0, Mozilla 1.1, and Opera 7 (8 choices). The application should work with the plug-ins RealPlayer, MediaPlayer and no plug-in (3 choices). The client OS can be one of the following: Windows 95, 98, ME, NT, 2000 and XP (6 choices). The server could be running IIS, Apache and WebLogic (3 choices) and Windows NT, 2000, and Linux (3 choices). In total, there are 1,296 combinations that would have to be run to completely test the software.

Step 3 is to find an orthogonal array that fits the problem. The smallest array that would fit this problem is an $L_{64}(8^2 4^3)$, where 64 is the number of rows in total, 8^2 means that there can be two variables that maximum can take eight different values each, and 4^3 means that there can be three variables that maximum can take four different values each. In this case, you would get an array that is bigger than necessary, but there is only fixed sized arrays so you would have to find the smallest that fits, in this case the $L_{64}(8^2 4^3)$. Which orthogonal array to use can be found in different books or on the Internet.

Step 4 is to map the problem onto the cells in the orthogonal array. Having done that for this example, could leave us with the array in table ”Table 4.3: An example of an filled orthogonal array.” on page 24. The empty cells can be filled with any of the other values in the same column, they are empty because the orthogonal array used did not fit exactly.

Step 5 is to construct one test case for each row in the orthogonal array. The values in the cells for the row is used as input to the system and the expected output comes from the specification for the system.

Table 4.3 An example of an filled orthogonal array.

	Browser	**Plug-in**	**Client OS**	**Web server**	**Server OS**
1	IE 5.0	None	Win 95	IIS	Win NT
2	IE 5.0		Win ME		
3	IE 5.0		Win 98		
4	IE 5.0	None	Win NT	IIS	Win NT
5	IE 5.0	MPlayer	Win 2000	WebLogic	Linux
6	IE 5.0	RealPlayer		Apache	Win 2000
7	IE 5.0	RealPlayer	Win XP	Apache	Win 2000
8	IE 5.0	MPlayer		WebLogic	Linux
9	IE 6.0		Win 95	WebLogic	Linux
10	IE 6.0	None	Win ME	Apache	Win 2000
11	IE 6.0	None	Win 98	Apache	Win 2000
12	IE 6.0		Win NT	WebLogic	Linux
13	IE 6.0	RealPlayer	Win 2000	IIS	Win NT
14	IE 6.0	MPlayer			
15	IE 6.0	MPlayer	Win XP		
16	IE 6.0	RealPlayer			Win NT
17	IE 5.5	MPlayer	Win 95	Apache	Win NT
18	IE 5.5	RealPlayer	Win ME	WebLogic	
19	IE 5.5	RealPlayer	Win 98	WebLogic	
20	IE 5.5	MPlayer	Win NT	Apache	Win NT
21	IE 5.5	None	Win 2000		Linux
22	IE 5.5			IIS	Win 2000
23	IE 5.5		Win XP	IIS	Win 2000
24	IE 5.5	None			Linux
25	Netscape 6.0	RealPlayer	Win 95		Linux
26	Netscape 6.0	MPlayer	Win ME	IIS	Win 2000
27	Netscape 6.0	MPlayer	Win 98	IIS	Win 2000
28	Netscape 6.0	RealPlayer	Win NT		Linux
29	Netscape 6.0		Win 2000	Apache	Win NT
30	Netscape 6.0	None		WebLogic	
31	Netscape 6.0	None	Win XP	WebLogic	
32	Netscape 6.0			Apache	Win NT

	Browser	**Plug-in**	**Client OS**	**Web server**	**Server OS**
33	Netscape 6.1	RealPlayer	Win 95		Win 2000
34	Netscape 6.1	MPlayer	Win ME	IIS	Linux
35	Netscape 6.1	MPlayer	Win 98	IIS	Linux
36	Netscape 6.1	RealPlayer	Win NT		Win 2000
37	Netscape 6.1		Win 2000	Apache	
38	Netscape 6.1	None		WebLogic	Win NT
39	Netscape 6.1	None	Win XP	WebLogic	Win NT
40	Netscape 6.1			Apache	
41	Mozilla 1.1	MPlayer	Win 95	Apache	
42	Mozilla 1.1	RealPlayer	Win ME	WebLogic	Win NT
43	Mozilla 1.1	RealPlayer	Win 98	WebLogic	Win NT
44	Mozilla 1.1	MPlayer	Win NT	Apache	
45	Mozilla 1.1	None	Win 2000		Win 2000
46	Mozilla 1.1			IIS	Linux
47	Mozilla 1.1		Win XP	IIS	Linux
48	Mozilla 1.1	None			Win 2000
49	Netscape 7.0		Win 95	WebLogic	Win 2000
50	Netscape 7.0	None	Win ME	Apache	Linux
51	Netscape 7.0	None	Win 98	Apache	Linux
52	Netscape 7.0		Win NT	WebLogic	Win 2000
53	Netscape 7.0	RealPlayer	Win 2000		
54	Netscape 7.0	MPlayer			Win NT
55	Netscape 7.0	MPlayer	Win XP		Win NT
56	Netscape 7.0	RealPlayer		IIS	
57	Opera 7	None	Win 95	IIS	
58	Opera 7		Win ME		Win NT
59	Opera 7		Win 98		Win NT
60	Opera 7	None	Win NT	IIS	
61	Opera 7	MPlayer	Win 2000	WebLogic	Win 2000
62	Opera 7	RealPlayer		Apache	Linux
63	Opera 7	RealPlayer	Win XP	Apache	Linux
64	Opera 7	MPlayer		WebLogic	Win 2000

PART II

AUTOMATED SOFTWARE TESTING

Automated software testing is a well established phenomenon, that has been introduced in many different systems and test processes before the work of this master thesis. That means that by studying previous work within the area many lessons have been learned before automating the process. Lessons like what the benefits and drawbacks within automated testing can be, have been learned and are presented in this part. Different methods and ways of how to automate a test process are also described. At the end of the part is a chapter about automated testing in embedded systems that is a base for what can be unique when testing embedded systems.

CHAPTER 6

AUTOMATING A MANUAL TEST PROCEDURE

Introducing automated testing in an organization can be an excellent way to raise the quality of the software of the company. One often hear stories that tells how people saved time with automated testing or that the tasks for the test people become more fun. But it is not allways easy to go from manual testing to automated testing. How well the procedure can be introduced depends on many factors. For example, how much does the software under test change between different releases?

6.1 Deciding when to automate

There are a number of factors that advocates for when to automate a manual test procedure and when not to automate. Situations in which one is adviced against to automate are easier to find in literature than the opposite. Six of the of the situations when one is warned from automating are [Fewster 1999]:

- The manual tests are seldomly run.
- The test object is volatile.
- The testing is easily humanly performed and hard to automate.
- Test object is an embedded system or a real-time system.
- A part of the test is physical interaction.
- The manually testing process is bad.

If the tests are seldom run, it will probably take a long time until the automated test procedure results in time savings. For instance, a test could be manually run once every month and every test execution lasts 15 minutes. To automate that process it might require ten hours or 600 minutes. That means that from a time perspective one will benefit from the automated test first after three years and four months.

When the test object constantly goes through major changes, it is most likely that the test automation also needs to go through major changes. When changes are time demanding so that it takes as long as, or longer to maintain the test automation than testing manually is then a common problem that the test automation will be abandoned.

In situations where the execution of tests are easily manually performed and hard to automate one is discouraged from automating the procedure. This situation occur for instance when a part of the test is to estethical appeal, easy for a human to decide but hard to automate.

In cases when the test object is an embedded system or a real-time system one is advised to rethink if one should automate because the system can require specialised tools or features that can be hard to implement.

Situations where physical interaction is involved, are situations when one is recommended not to automate the test procedure, since physical interaction can be hard to automate. It can fo instance be hard to automate turning on/off power, unlock a lock with a key or load a cd into a cd-player.

One important factor that should be considered before deciding if one should automate testing is how good the manual test procedure is. It is advised not to automate if the manual testing process has problems as for instance that it is badly defined. This is expressed by Mark Fewster and Dorothy Graham with: “Automating chaos just gives faster chaos” [Fewster 1999, p.11].

Mark Fewster and Dorothy Graham also describe characteristics that are positive when introducing automated testing. Characteristics as: important test, can easily be automated, gives pay back quick and test is being run often are factors that are good when automating [Fewster 1999]. As mentioned earlier, maintenance can be a problem for automated testing. Regression testing can therefore be specially

suitable for automated testing since the maintenance is low due to the repeatedly running of tests.

6.2 Creating test cases for automated testing

The techniques that are described in “4 Techniques for creating test cases” on page 17 are all applicable also for creating automated test cases.

What differs the creation for automated testing from manually testing is that the test cases must be more specific declared in every way. The need for having specified test cases is based on the automated test being performed by a machine that only can process what it is told to. It is therefore required that every detailed is well specified to be able to get the execution of test case that is intended.

6.3 Test performance

There exist a wide range of techniques that are described in various forms, in this chapter are different scripting techniques described. The main reason for describing the scripting techniques is that the base of most automated test performance origins from it.

6.3.1 Scripting techniques

If one records a test case being performed manually it results in one linear script that can be used to replay the actions performed by the manual tester. Doing this for several test cases will result in one script per test case. The script is really a form of program, a set of instructions for the test tool to act upon. Having one such script for each test case is not really efficient since many test cases share common actions, like “login in to the webpage”, “open a file”, etc. Therefore, most automated test tools come with a test script language that will help you in creating efficient and manageable test cases. There is five types of scripting techniques that can, and most likely will, be used together [Fewster 1999]:

- linear scripts;
- structured scripting;

- shared scripts;
- data-driven scripts;
- keyword-driven scripts.

6.3.1.1 Linear scripts

A linear script is what one gets when one records all actions in a test case that is performed manually. This script can later be used to playback the exact same inputs from the manual script. With this technique, it is easy to create test cases, one can quickly start automating and the user does not need to be a programmer. It is also good to use when one is demonstrating the software for a customer, since one know exactly what is going to happen, and there will be no unknown events because one is nervous and input some false value in a text box. On the other hand, it is not a very good procedure when one should automate a very large number of test cases, since it typically takes two to ten times more time to produce a working test script using linear scripts than to manually run the same test [Fewster 1999]. Every script also needs to be created from scratch and they are vulnerable to changes in the software under test and hence have a high maintenance cost.

6.3.1.2 Structured scripting

Structured scripting is based on special instructions to control the execution of the test. The special instructions can either be of control type or of calling type. The calling instructions invokes other parts of the script and makes the reuse of code possible. The control instructions can be divided into three subgroups, the sequence, decision and iterator. The sequence control is based on the same principles as described in “Linear scripts” on page 36, the events are sequentialy being executed. The decision control makes it possible to as the name implies have decisions in the script, as for instance if statements. The last one iterator control allows iteration over sequences in the script. The largest advantage with structured scripting is that the test execution can be robust and use the instructions to control the execution to discover events that may occur in the test object. [Fewster 1999]

6.3.1.3 Shared scripts

Shared scripting has functionalities that are common for several test cases been lift out of the script belonging to the test case and gathered in an additional script. This way is the most maintenance friendly so far presented, even though it may include maintenance problems. If the functionalities that are in separate scripts are hard to find, the test script creator will probably create a new script for the same purpose which may lead to maintenance problem. Another disadvantage that affect the maintenance is that it requires one script per test case. The shared scripting techniques is a good way to automate test when it is a small system that should be tested or if there is a small part of a larger system that should be tested. [Fewster 1999]

6.3.1.4 Data-driven scripts

Data driven scripts are based on the structure to have all inputs for the tests stored in a separate file instead of in the script. The script does only contain functionalities, that means that it is not necessary to have one script per test case. This technique offers advantages as: the effort of adding tests with similar actions that have been previously executed is low and tests can be created and added by creators without programming experience. Disadvantage with data-driven scripts is that the initial set-up as constructing the structure of data files, designing and implementing the script can be demanding. [Fewster 1999]

6.3.1.5 Keyword-driven scripts

The keyword-driven scripting method is an extension of the technique described in “Data-driven scripts” on page 37. Instead of only having test inputs separated from the script that contains the functionalities to be executed, is also keywords that are used to describe what events that should be executed separated from the functionalities. The test case now holds what to test in form of a list of keywords to execute and a control script is added to the test tool. The control script is responsible for going through all test cases and controls the execution by for example open the program under test and then for each key-word in the test cases, calling a supporting script that holds the real implementation of the key-word. [Fewster 1999]

6.4 Test evaluation

The automated test evaluation can also be referred to as the verification and is based on what the test case creator state is the expected outcome and the actual outcome of a test. To be able to verify the output, comparisons are used. The focus of this part is to declare concept that are important in an automated test evaluation.

6.4.1 Simple and complex comparison

A simple comparison is when the comparison is an exact comparison of the expected, where no differences are allowed in the output. In the opposing, complex comparison, are known differences taken into account in the evaluation. Simple comparison is in some literature referred to as dumb comparison and complex comparison is referred to as intelligent comparison. [Fewster 1999]

6.4.2 Sensitivity of test

The sensitivity of the test describes how much comparison that should be done. The case when there is being executed as many as possible comparisons in the verification, it is described to be a sensitive test. Positive effects by doing so is that more changes in the output are likely the be discovered. Negative causes are that this type of evaluation often requires high maintenance. If one compares a minimum quantity, it is referred to as robust testing. One of the two options should be decided when designing the automated test evaluation. Sensitive testing is recommended when designing test automation for higher level testing, and robust testing is recommended when constructing automated test at a more detailed level where focus is at specific aspects instead of having a breadth testing. [Fewster 1999]

6.4.3 Dynamic comparison

When comparisons are being executed during the performance of the test it is referred to as dynamic comparison and it is the most commonly used comparison techniques in commercial tools. By using

dynamic comparison a intelligent system can be created where different actions are executed depending on the result in the comparison, it also makes it possible to fetch errors during performance and end the performance or make another attempt to execute the action that failed. For instance if a step in execution the test is to connect to a database and it fails, it could be of interest to catch that event and try again. The drawback with dynamic comparison is that since it requires verifications built into the software that executes the test, it is complex and high complexity can result in high maintenance costs.[Fewster 1999]

6.4.4 Post-execution comparison

Post-execution comparison is when all comparisons are executed at the end of the run of the test case. This type of comparison does not offer the positive effects that are given in the dynamic comparison, but it results in other benefits that should be taken into account when designing the automated test. This way of comparison makes it possible to group the result which can be used to compare the most important parts first and if the comparison results in failed, no more comparison needs to be executed. If one stores not only the outcome when the whole test case is executed it renders possible to evaluate the state during the execution of the test, which makes it possible to achieve some of the information that is accessible in dynamic comparison. The largest benefit with post-execution comparison is that is often not as complex as dynamic comparison, which gives lower maintenance costs. [Fewster 1999]

6.5 Test result

The test report is the common way where the result of the test execution is stored. In the [IEEE 1998] is a standard defined on what the contents of a test report should be. Noticeable is that the standard is not made for automated testing but for manual software testing, which results in that not all that is declared in the standard is applicable in automated test. The standard declares three different documents that should be created as parts of the test report; the test log, test incident report and the test summary report.

The test log is as the name implies used to be able to log what have occurred during the execution of the test. It should contain: a test log identifier, description of the execution and the activity and event entries that have occurred during the test.

The test incident report is used to be able to store information about unexpected events during test execution. It should contain: a test incident report id, summary of the execution, incident description and impact of the incident.

The test summary report is used to sum up the test execution and presents any incidents that may have occurred during execution and the result of the test.

CHAPTER 7

AUTOMATED TESTING IN EMBEDDED SYSTEMS

The electronic devices used in every day life such as wash machines, cell phones, PDAs and car stereos are all more and more used through out the society. The computer system in such a device is called an embedded system.

7.1 Definition of an embedded system

The term embedded system is not a standardized global term that refer to the exact same thing all over the globe. However, there exist some features that are more or less common to all embedded systems [Karlsson 2006]:

- They are part of a larger system (host system), therefore the term *embedded* is used. There is an interaction between the embedded system and the host system, which are carried out frequently or continuously.
- Embedded system is not intended to have a general functionality and are not supposed to be re-programmable by the end-users. Once a embedded system is taken into use, its functionality is generally not changed throughout its life time. For example an ATM which normally only can be used to withdraw money from and show some information about the users account, will probably never be reprogrammed to function as a cruise control system in a car. But a desktop computer, on the other hand, has a very general intention of use.

- Embedded systems have real-time behaviour. They must, for the most, react to their environment in a timely manner.
- They consist of both hardware and software components. The hardware of a desktop computer has to be generally designed to be able to be used for as many tasks as possible. This leads to a risk of wasting resources. For an embedded system, however, the range of use is known at design-time, which leads to that they can be specifically designed for running only the known applications leading to the best performance at minimal cost.

7.2 Embedded software vs. regular software

It is very different testing an embedded system software automatically from testing a normal software running on a desktop computer. When testing a normal software, there are normally no problem in inputting data and reading responses from the software, since the software normally uses the keyboard, the mouse, a database or a file to get input and the monitor, a printer, a file or a database to display outputs. If the program are using a file or a database for reading and writing input and output, it is rather easy to write a script producing inputs and reading outputs and later evaluates the result. On the other hand, if the program uses the keyboard, the mouse and the monitor for input and output, it all comes down to common automated GUI testing, which there exist a lot of tools to perform. The automated GUI test systems come with a test recorder which can record the test scripts that are written or which can record mouse clicks and key strokes. These recorded test cases can later be play backed as simulated user interactions. There is also a result recorder which records what happens on the screen when input is given to the system. The big difference between regular automated testing and automated testing in embedded systems is the interface to the system. On a normal desktop computer, this interface is provided by the operating system. In an embedded system however, there is no regular OS, rather some kind of RTOS, but the interfaces are different and dependent on the hardware [Kandler 2000]. An embedded system can have interfaces to the environment like a GSM transceiver in a cell phone, a touch pad in a PDA, or signal output ports on a system controlling a manufacturing process or in an embedded system for a nuclear power

plant. To interact with the system over these interfaces, special software may have to be developed that simulates the devices that sends and receives signals to and from the embedded system software.

7.3 Defining the interfaces

There exist no standard API for sending data back and forth within embedded systems. Eventually, one will end up having to hook into the hardware at some point or to build in some kind of customized interface in the application. The point where the interface is built, either at hardware-level connected directly on the processor or some kind of a serial connector, or at software-level built in the application, depends on what testing one wants to do. If the interfaces are made completely surrounding the application, it should be possible to carry out function testing of the system as whole. But if an interface is implemented for example between the module handling calculations and the module updating the display in a pocket calculator, function testing on the calculation module and the modules below can be done, but the display module cannot be tested. This is weighing that has to be done when defining the interfaces. It could for example be too expensive or complicated to hook into the connectors to the hardware devices surrounding your software.

7.4 Signal simulation

Once determined which interfaces exist, one start thinking on which signals are running through the defined interfaces. The signal simulation can be divided into two sub categories: full simulation and switched simulation. [Kandler 2000]

7.4.1 Full simulation

If all hardware that sends input to and reads output from the embedded system is simulated, the term full simulation is used. [Kandler 2000]

7.4.2 Switched simulation

Another way of simulating input signals is to only simulate some signals. For example by interrupting the actual signal and to provide a way to modify it before reaching the embedded software. In some systems, this may be a better way because the input signal could be too complicated to generate. With this approach one have the ability to generate errors in the system by corrupting the real-time signals coming into the system. Another reason for choosing this approach is if the system has only some signals that needs to be controlled in order to run the tests, the other ones can be left untouched. [Kandler 2000]

CHAPTER 5

AUTOMATED TESTING

Automated testing raises many different opinions, some of them are positive and some the opposite. When implementing automated testing both sides have to be considered to be able to succeed, they are therefore presented in this part.

5.1 Introduction

According to Fewster and Graham [Fewster 1999], there are four attributes which distinguish how good a test case is. The first is the defect detection effectiveness of the test case, i.e. if it at all is able to detect the error it was designed to detect. The second quality is that one would want the test case to be as exemplary as possibly. An exemplary test case will test many test conditions in a single test run. The last two are cost factors that affect the quality of the test case: how economic the test case is to perform, analyse and debug, and how evolvable it is, i.e. how much effort is needed to adopt the test case to changes in the software under test. It is often hard to design test cases that are good according to all these measures. For example, if a test case can test many conditions it is likely to be less economic to analyse and debug and it may be require big adoptions to software changes.

When a test case is automated, its value on the economic scale is most likely to rise if it is to be performed several times. However, if the test case tends to be run only occasionally it tends to fall since the development cost of the test case is higher for an automated test than it is for the same manual test. An automated test also tends to be less evolvable than its manual correspondence [Fewster 1999].

Automated testing is well suited to be used as regression test tools on already existing software since a test that is going to be run automatically has to be run manually first. This is because the test case itself has to be tested so it finds the defects it is intended to and do not hide flaws in the software instead of bringing them up to the surface. Therefore, these testing tools are not “testing” tools, rather more like “re-testing” tools, i.e. regression testing tools.

5.2 Benefits of automated testing

Automated testing demands less manpower to be executed than manual testing does. The role of the tester during the test execution with manual testing is to set up and input data into the system and check the actual output according to the expected output. With automated testing, the role of the tester is to start the system and to tell the automated testing software which test cases to run. The actual execution is done by the automated testing software itself, which gives the possibility to run a test case more times or more test cases to be run than when manual testing is used. This frees up the tester during the execution who then can devote herself to other tasks.

5.3 Drawbacks of automated testing

Not all tests are suitable for automatic testing. Exploratory or lateral testing is better to be performed manual. If the software under test is not yet stable, the errors would be found very quickly with manual exploratory testing.

Manual testing finds more errors in software than automatic testing does. It is more likely that an error is revealed at the first test run and since a test case has to be verified manual, it most likely that the error is found by manual testing. In fact 85% of all found errors are found by manual testing. [Fewster 1999]

A testing tool can only detect differences between expected output and real output, it can not interpret the meaning of the output. This leads to that more work is demanded by humans during the evaluation of the test cases than is needed when tests are run manually, since every expected output must be more carefully verified [Fewster 1999].

A tool does not possess any imagination. If the test case has errors, those errors can be found and corrected immediately during the test run be a human, which leads to the test can be carried out successfully anyway.

PART III

IMPLEMENTING AUTOMATED TESTING

IVU Traffic Technologies AG in Aachen, Germany, offers solutions to the public transport sector both nationally and internationally. The implementation of automated testing was performed within one of their projects. The goal was to automate a part of their system testing for one of their products, the i.box. The i.box is an on board computer that is used in public transport vehicles, and it offers a wide range of functionalities. One of those is the positioning of the vehicle, which is for instance used to support the driver in following the correct timetable. The chapters in this part describe how the manual functional testing process for the positioning was transformed into automated testing, with steps as constructing test cases, performing tests, evaluating output and generating test report.

CHAPTER 8

DESCRIPTION OF THE SYSTEM

Today's public transport companies use numerous technical systems, where an on-board computer, supporting the driver and the public transport command centre is one of them. The system that was in focus during the automatic testing in this study is such a system, it is called i.box and is a special implementation of a so-called IBIS.

8.1 IBIS

An IBIS (Integrated on-Board Information System) is a general term for an on-board computer used in public transport vehicles for controlling all computer-aided service functions on the vehicle. The IBIS controls for example passenger systems like electronic displays, automatic announcements of stops and printing tickets. On a bus, it also controls traffic lights and changes to green when the bus approaches the light. Through audio and text messages, the command centre has the ability to communicate with the driver via the IBIS. For more information (in German) about IBIS, see [VöV 1984], [VöV 1987] and [VöV 1991].

8.2 The i.box

The i.box is an embedded real-time system, developed by IVU Traffic Technologies AG, that is used on buses and trams during the trip. Among other functions, it has a capability to show the current position of the vehicle along the journey, the next stop along the route, and the time in relation to the timetable. It is also used for ticketing, for controlling the periphial devices on the bus and the tram, and to

inform the passengers via information boards and speaker systems inside and outside the vehicle. Another important functionality that has been a part since the beginning of on-board computers, is the possibility to communicate with the command centre of the inter urban traffic company. This enables the centre to send messages and talk to the driver over radio communication during the journey. It also enable the sending of emergency messages. The command centre does also automatically get the current position of the vehicle and time in relation to the timetable. To be able to give the desired information to the users of the system, timetables is at regular intervals transferred over WLAN to the i.box when the bus or the tram is at the depot.

8.2.1 Positioning

The positioning is a very important module in the i.box to allow the users to utilize the above mentioned functions. The task of this module is to calculate the current position of the vehicle. To perform this task, the i.box uses input data from four different sources:

- a GPS-receiver;
- a distance pulse;
- sensors giving signals if the doors are opened or closed;
- the driver through a keyboard or a touch screen.

8.2.1.1 GPS-receiver

The vehicle has a GPS-receiver that every second sends a GPS message to the GPS module in the i.box. The GPS messages that are sent is of two different types, namely GPGGA and GPRMC messages. Both contain the current GPS position along with some other information. This information is in the GPGGA message for example time, number of satellites used in the calculation of the position, altitude, and data for the geoid and ellipsoid used in the calculations. For the GPRMS this information is among other speed, date and time. The date and time is used by the i.box to adjust its clock periodically.

8.2.1.2 Distance pulse

Like the trip meter (mileometer) measures covered distance of a car, the i.box also measures the covered distance of the bus or tram by using a sensor and a transmitter. When the sensor recognise that the wheel has been turned, an impulse is sent to the i.box. The i.box is configurated with how many impulses that is responding to a meter. Number of impulses that corresponds to a meter is calculated by knowing the size of the wheel. Normally, the distance pulse transmitter sends four impulses per meter in a bus and seven impulses per meter in a tram. The counting of distance pulses is reset every time the vehicle reaches a stop, and starts counting when leaving that stop. The covered distance is cyclically compared to the distance between stops read from the travel plan and taken into account when calculating where along the scheduled route the vehicle is.

8.2.1.3 Door sensor

When the i.box calculates the position, it also considers if the doors of the vehicle is closed or opened. The positioning algorithm in the i.box uses variables like how many times the doors have been opened since the last recognition of a stop and with which interval they were opened.

8.2.1.4 Keyboard and touch screen

The i.box is equipped with a keyboard and/or a touch screen with which the driver can enter information to the system about the position. If the driver enters a position on the route that differs from the one the system has calculated, then the position that the driver inputs is always considered the correct one.

8.2.1.5 The positioning algorithm

The i.box uses several parameters for deciding where the vehicle is on the route. They are independent of each other, for example the distance pulse meter counts only the covered distance since the last stop, but the GPS on the other hand, measures exactly where the vehicle is on the earth and they do it whether the other system works or not. These two systems are used together, along with signals from the doors and input from the driver. The observant reader may notice that four different systems are used to perform the same thing. The

reason for not only having one system is that one wants the effect that is reachable by using all of them. If one look into the four systems starting with the GPS, one sees that the accuracy of only using GPS is not sufficient in all situations. GPS can therefore not be used alone. Looking into the calculations based on the distance pulse it can neither be used alone, since the calculations cannot be exact since the distance varies when driving the same route different times. Because stops can be of different size, i.e. they can be designed to allow one ore many buses to stop at the same time, there is defined areas for the distance pulse meter and the GPS within which the i.box should synchronise, i.e. set the position to a stop. There exist three different synchronisation areas: GPS area, distance pulse area before the stop, and distance pulse area after the stop, see *"Figure 8.1: The three different areas that are important for synchronisation." on page 51*. The GPS area is used all the time, the distance pulse area before the stop is used when the vehicle is approaching the stop and the distance pulse are before the stop is used when the vehicle is leaving the stop. The distance meter synchronisation areas can be set different for different stops and the GPS area is setup for each type of i.box in a configuration file.

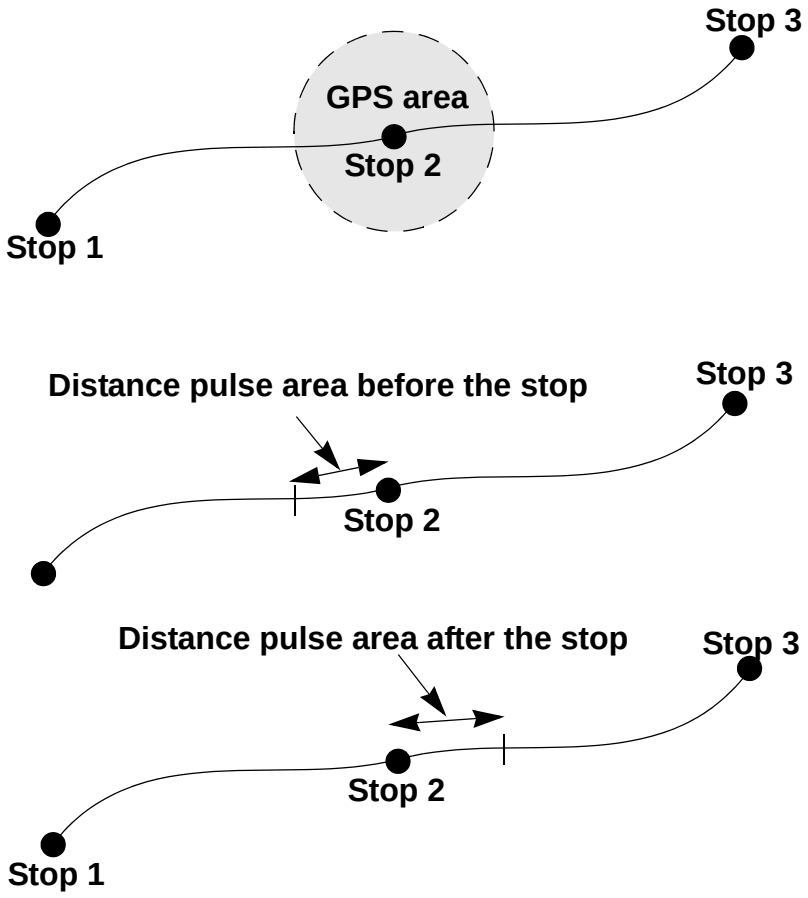


Figure 8.1: The three different areas that are important for synchronisation.

8.2.2 Characteristics suitable for test automation

When the automation process is concentrated around the positioning functionality, it appears that the manual testing process of this part fulfils some of the requirements that is described in *"6.1 Deciding when to automate" on page 33*. The first part that coincide for automatisation is that it is a question of regression testing, a number of tests will unchanged be carried out to make sure that no outer chang-

es have affected the localisation functionality. Outer changes could be that the customer has desires for a customized product where functionalities have been added or removed, or changes like for example that the software will be used on another type of hardware. In both cases, no changes has been made to the positioning functionality, but there is still a need to test that it continues to work as expected.

Another quality that according to previously mentioned section speaks in favour for automation of the test process is that changes in the positioning functionality is unusual, which results in no regular changes in the automation process should be necessary.

8.2.3 Characteristics not suitable for test automation

That the positioning functionality remains unchanged when different system solutions are developed for different customers and used on different hardware platforms, are mentioned as positive for automation of tests. This also gives qualities that are negative during test automation. The fact is, that some parts of testing are different for different customers and that their system is dependent on their wishes. As described in *"6.1 Deciding when to automate" on page 33*, it is undesirable to apply automated testing when tests are different between different realizations. Because if the time needed for administration of automated test is longer or as long as running the tests manually, the automated process will probably not be used. For example, one difference between different implementations for different customers is that some customers want their employees to login to the system before entering which line they will drive. Some customers does not have this login dialogue, so on that systems it is possible to enter the line number directly. These differences in the implementation leads to the manual testing being carried out differently for the different systems.

CHAPTER 9

AUTOMATING THE TEST PROCESS

The automated test procedure was divided into four different steps: to construct test cases, to execute the test, to execute an evaluation of the result and to generate result.

The first part, to construct test cases for automated testing, was performed by hand based on the test cases that existed for manual testing and on the methods described in “4 Techniques for creating test cases” on page 17. Since the test procedure deals with regression testing, which leads to that new test cases are not created often, there is no need for a tool to generate test cases.

The last three parts of the process to automate testing was performed by specifying requirements on the different parts, doing an survey of some of the tools that where available, and after evaluating the result of the investigation deciding to develop tools that fulfilled the specified requirements.

9.1 Test method

In chapter “6 Automating a manual test procedure” on page 33 different test methods have been described. This section in the test case aim to present what methods that have been used and why those have been used. The modifications of the methods that have been made are also described, whenever there exist any.

9.1.1 Full signal simulation vs. switched signal simulation

As said in “7 Signal simulation” on page 43 there exist two types of signal simulation. Only the signals that has anything to do with positioning, i.e. GPS, distance pulse, door signals and keyboard has been simulated, the others have been left untouched, so switched simulation has been used.

9.1.2 Scripting technique

A set of key-words to use when writing the test cases have been defined, and are in this thesis referred to as the actions in the test case. A control script along with a support script for each action have been implemented, see “9.4.3.4 Test case execution” on page 74.

9.1.3 Dynamic comparison vs. post execution comparison

When testing the i.box it is possible, when executing some actions, to get feedback that makes it possible to measure that the action has been executed in the i.box. For instance, it is possible to see that a button has been pushed in the i.box by reading outputs from Auster (See “9.4.1 Auster” on page 68 for more information about Auster). But unfortunately, it is for all actions not possible to get feedback from the i.box when the action has been processed. That rise problems when trying to use dynamic comparison.

The reason for choosing post execution comparison over dynamic comparison is that it simplifies the development of the automated test tool. Both test methods are possible to use but if post execution comparison is used, the two parts can be clearly separated and developed individually. If dynamic execution was used, the automated test evaluation tool was forced to be integrated as a part of the automated test performance tool.

9.1.4 Simple and complex comparison

A mixture of simple and complex comparison has been implemented in the evaluation. Some of the expected outputs are better fit to be evaluated with simple comparison and does not require the use of complex comparison, where on the other hand some expected outputs need to be evaluated with complex comparison.

9.2 Constructing test cases for automated testing

A selection of test cases for manually testing was the foundation of what tests the automated system should be able to execute. This set of test cases was analysed and a new set of test cases was constructed with the methods described in “4 Techniques for creating test cases” on page 17. This selection was the test cases that needed to be translated into test cases that could be run as automated tests. The model of an automated test case was constructed outgoing from [IEEE 1990] and consist of the elements: administrative information, preconditions, actions and expected values. An example of a manual test case that has been transformed into an automated test case can be found in the figures ”Figure 9.2: An example of a test case for manual testing.” on page 56 and ”Figure 9.3: An example of a test case for automatic testing.” on page 57

SYNCHRONIZATION AT THE START NET POINT

That the start-net point of a trip is reached is determined by the following critera:

1) *The received GPS coordinates answer to (with a maximal divergence of a setup parameter) the GPS coordinates of the GPS coordinates of the start-net point.*
2) *The door is opened within the stop area of the start-net point.*
3) *The driver confirms the start net point manually.*
4) *A new trip is chosen by the driver.*

Test case 1.1.3/1:	Chosing a valid trip. The in setup parameter 5211 *GPS_Standardfangbereich* defined tolerance area (20 m) is fulfilled.
Expected output:	The start net point is automatically set in the driver's terminal.
Actual output:	

Function correct. ☐
Function has errors ☐
Function not useable ☐

Figure 9.2: An example of a test case for manual testing.

Name	Synchronisation at first net point		
Test case id	1		
Description	Navigation with GPS, automatic synchronisation inside GPS area		
Author	Johan and Katrin	**Kurs**	03
Version	1.1	**Trip**	17
Creation time	2007-03-07 09:14:43	**Block**	200003
Last updated	2007-04-07 12:31:25	**Time & date**	2007-05-07 14:24:45
Bereich	MB010	**GPS Status**	on
Line	725	**Distance pulse status**	on
Actions	Push buttons	buttons: confirm_trip	
	Set initial position	gps_x: 43306549 gps_y: 515110000	
	Drive to a point based on the GPS area	gpsarea_dist: -1 stop_point: 53280270	
Expected values	The vehicle is at a net point	Last net point: 53280270 Distance since last net point: 0	
	The status of the trip of the vehicle	Trip status: 1	
	Synchronisation status of the vehicle	Distance since last synchronisation = synchronized	

Figure 9.3: An example of a test case for automatic testing.

9.2.1 Administrative information

To be able to easier administrate the test cases and to ease readability some additional information that is not used during the test case execution was added: a textual name of the test case, a description of what the test case is supposed to test, the author of the test case, a version number, creation date and time for the test case, and the date and time for the last update of the test case. The last two information fields are automatically created and updated by the system.

9.2.2 Preconditions

As when testing manually it is a must that all preconditions that should be in force are stated in each test case. In general when constructing test cases, the preconditions normally describes what software it should be able to run on, or which requirements that exists on the hardware. Since one thing that one wants to test is that same correct behaviour exists in different software and or hardware environments, the preconditions in this study are not the preconditions that one normally expects. There are no requirement on the hardware or the software, besides that it should be an environment that at least one i.box is aimed to be working on. Instead the preconditions gives information about the settings that needs to be made so all different i.boxes are in the same state when performing the test case. The only preconditions that describe functionalities of the surrounding environment are conditions that describe the status of the GPS and distance pulse. Altogether, the preconditions that must exist in the test case are the following: GPS status, distance pulse status, time, domain, base version, line, course, trip and block.

9.2.2.1 GPS and distance pulse status

The GPS setting can be on or off, and describes whether a GPS signal should be sent to the i.box. Like the GPS setting, can also the distance pulse status be on or off. It is used to indicate if distance pulses should be sent.

9.2.2.2 Time

The time precondition is one of the settings that are needed to set all i.boxes in the same state during test execution. Depending on the time settings, different timetables will be shown in the GUI of the i.box. Therefore, the time needs to be set so the actions that are executed coincide with how the actions were meant to be executed by the test case creator.

9.2.2.3 Domain and base version

The domain and base version are needed for fetching the correct timetable from the database.

9.2.2.4 Line, course, trip and block

Line, course, trip and block (German: linie, kurs, fahrt and umlauf) are, as time setting, used for setting the i.box in same state during execution. A difference between the time setting and the others is that only the former is used in every test execution. Depending on what software the i.box is running, different combinations of the preconditions will be used. The trip precondition is also used when fetching the timetable from the database.

9.2.3 Actions

The actions describe the course of events that should be executed before the output should be read. The biggest difference between these actions and the actions in test cases for manually testing is that both the actions and the parameters that belong to the actions are more specifically specified. In order to automatically perform the events that are described in the test cases for manually testing, the following fourteen actions have been created.

9.2.3.1 Push button

This action is used for simulating button pushes at the keyboard or the touch screen of the i.box. As parameters it can take one or a several buttons that should be pushed. Since different i.boxes have different GUI representations it is not sufficient with only the possibility to push specific buttons. If one pushes button one, and then button two in one i.box the state of the i.box will most likely be

different from the state another i.box would be in after the same sequence of button pushes. Since one of the requirements of the system is that different systems should be able to be tested with the same test case, one must have the possibility to also specify what one would like to do with the button pushes. Therefore, it is also possible to have a string as a parameter to this action. This string describes what should be performed and it corresponds to different button pushes in different i.boxes. For instance “confirm trip” answers to button sequence 1 2 3 in one i.box and button sequence 7 2 8 in another.

9.2.3.2 Control the door

As the headline implies, this action is used for simulating the signal that comes from sensors that are placed at the door of the vehicle. By given arguments as open or close the i.box will get signals that represent door opening or closings.

9.2.3.3 Wait

The wait action makes the test system to pause in the execution the number of seconds that is given as argument. It can be used to give the i.box more time to execute its commands and it is also useful if a test person wants to manually control something in the i.box during test execution.

9.2.3.4 Set position

This action is used to set a coordinate in the i.box and the test system. If the GPS is active the coordinates will be sent to the i.box, otherwise it will only be stored in the internal data of the test system. The action is mostly used in the beginning of the test case, in preparation of executing a drive action. It takes either GPS coordinates that should be the new position as arguments, or it takes an identifier of the stop point that should be the new position.

9.2.3.5 Change GPS status

The action that is used to change the status of the GPS and can take one or zero as argument. When the argument is one, the change GPS status action will set the GPS to active and start sending the GPS coordinates of the current position. If the current position of the vehicle has not been initialised, no GPS coordinates will be sent. On the oth-

er hand, if the argument is zero the internal GPS status will be set to not active, and if GPS coordinates are currently being sent the sending will cease.

9.2.3.6 Change distance pulse status

Sets the status of the distance pulse and can take the arguments one or zero. If the argument is one the status is set to active and if it is zero it is inactive. If the status is active, distance pulses will be sent if a drive action is executed, if it is inactive it will do the opposite.

9.2.3.7 Drive N meter

Simulates driving straight to the east a number of meters that is given as argument.

9.2.3.8 Drive to position

Simulates driving to a given position, which is described directly by GPS coordinates in DMS format as parameters to the function or by an identifier of the stop position that it should simulate driving to. If the GPS coordinates are given as arguments, it always drives the shortest way to the coordinate from the current position. When the stop point is given as an argument, it fetches the GPS coordinates for the given stop point from the database. The distance that should be driven is then depending on the current position of the vehicle. If the current position is at the stop point before the one given as the position to drive to, the distance to drive is then fetched from the database that contains information about different trips. That means that the simulation will simulate driving exactly the distance that is between the two stop points in the real world. In other cases it is not possible to know the distance the vehicle should be driving in the real world, therefore it will simulate driving the shortest distance to the GPS coordinates.

9.2.3.9 Drive to a position by driving a certain distance

This action is similar to the action “9.2.3.8 Drive to position” on page 61, but in this action the test case constructor also defines the distance that should be driven. It can take either GPS coordinates in DMS format or an identifier to the stop point as arguments. The difference between the two alternatives is that if the identifier is given

the GPS coordinates will be fetched from the database with information about the trip in the other case the GPS coordinates that are given as arguments will be used.

9.2.3.10 Drive to a position that is relative an area corresponding to a stop point

As mentioned in the description of the i.box, see “8 Description of the system” on page 47, there exist three areas that are of importance to the location functionalities: GPS area, area before reaching the stop and area when leaving the stop. Depending on where the vehicle is relative the different areas different events should be occurring in the i.box, this is the reason for having actions that can simulate driving relative the different areas. These actions are:

1. Drive to a position that is relative the GPS area at a given stop point
2. Drive to a position that is relative stop point area before stop point at a given stop point
3. Drive to a position that is relative stop point area after stop point at a given stop point

Action 1, 2 and 3 above requires two parameters: id of the stop point that the vehicle should drive to and the distance that should be between the area and the end point. To be able to test behaviours that require the vehicle's position to be relative two areas at the same time, these actions are necessary:

4. Drive to a position that is relative GPS area and relative stop point area before stop point at a given stop point
5. Drive to a position that is relative GPS area and relative stop point area after stop point at a given stop point

Action 4 and 5 above require three parameters: id of the stop point that the vehicle should drive to and the distances that should be between the two areas and the end point.

As in the other drive actions the distance to drive in these five actions is depending on the current position of the vehicle. If current position is at the stop point that is before the stop point given as argument, the distance to drive is fetched from a database that contains

information about different trips. In other cases it is not possible to know the distance that the vehicle should be driving in the real world. Therefore, it will simulate driving the shortest distance to the GPS coordinates.

It is possible to simulate driving to a position that is at, inside or outside the area. That is done by given zero, negative or positive value as the parameter value that describes the number of meters that should be between the area and the end position.

If the vehicle's current position is at the stop point that is given as parameter, in the action 3 and 5, the direction to drive is straight to the west and in the same situation in action 1,2 and 4 the direction to drive is straight to east.

When executing one of the actions that simulate driving relative the area after the stop point (action 3 or 5) the centre of the stop point is always passed.

9.2.4 Expected output

The different expected values that existed in the test cases for manually testing were translated into specified expected values. At least one expected value most be specified in the test case to have a complete test case. The name of the expected value describes what output is of interest. The expected values are:

1. Vehicle is at specified net point
2. Vehicle is between net points
3. The status of the trip
4. I.box has been synchronized
5. Next net point is at specified net point
6. Vehicle is off-route

An expected value could be having more than one thing that one is interested in to see the result for, for instance if one would like to check that the vehicle is between two net points, that means that one is testing what stop point that was the last one that was passed, which is the next one and how far has the vehicle driven since last stop point. To be able to test different things at one expected value, at

least one parameter were added to each expected value. What parameter values that needed to be added, was controlled by what could be read as output from the i.box. The expected values and their parameters are:

1. Vehicle is at specified net point
    1.1. Last net point
    1.2. Distance since last net point
2. Vehicle is between net points
    2.1. Last net point
    2.2. Next net point
    2.3. Distance since last net point
3. The status of the trip
    3.1. Trip status
4. I.box has been synchronized
    4.1. Distance since last synchronisation
5. Next net point is at specified net point
    5.1. Next net point
6. Vehicle is off-route
    6.1. Off-route status

Three parameter values have special conditions. The parameter value “Distance since last net point” to the expected value “Vehicle is at specified net point” should always be zero. That means that the test case user must not specify the parameter value when creating a test case with the expected value “Vehicle is at specified net point”. This is done automatically when storing the test case. Another parameter value with a special condition is the parameter “Distance since last net point” in the expected value “Vehicle is between net points”. The expected value should always be larger than zero. That means that it is not possible to set a value that is correct, since all values that are larger than zero means that the vehicle is between the two stop points. The last parameter value with a special condition is the pa-

rameter “Distance since last synchronization” in expected value “I.box has been synchronized”. The valid expected values to the parameter are “synchronized” or “not synchronized”. The actual value for a success for “synchronized” is always zero, but in the case when one is testing if it is not synchronized it is not possible to predict exactly what the expected value is. It should be considered to be a success if the actual value fulfils the condition that it is larger than zero. This has to be taken into account when executing the evaluation of the test, since it is not correct to perform a regular equal comparison.

9.3 Analysis of existing test tools

As mentioned in the chapter description, see “9 Automating the test process” on page 53, a brief analysis of existing test tools was performed in the search for finding a tool that could perform at least one of the following parts, execute test, evaluate the result and generate test report. The requirements on the different parts are described in “9.3.1 Requirement on tool” on page 65. Some of the tools that was analysed are briefly described under headlines with the name of the test tools.

9.3.1 Requirement on tool

Requirement that was common for all of the three parts (execute test, evaluate the result and generate test report) is:

- costs should be low.

  The tool for executing the tests should be:

- able to test embedded systems;
- compatible with the i.box, so it can read outputs and send inputs;
- able to simulate GPS coordinates in DM format and send it to the i.box;
- able to simulate distance pulses and send it to the i.box, alternatively be able to use existing test tool to send distance pulses, see “9.4.2 B4 test system” on page 70;

- able to simulate signals from door sensors and send it to the i.box, alternatively be able to use existing test tool to send signals, see “9.4.2 B4 test system” on page 70;
- able to send all the above mentioned input signals synchronized;
- able to read input data from oracle database, in order to fetch information about net points and timetables;
- able to simulate button pushes on the i.box;
- able to execute all the specified actions in “9.2.3 Actions” on page 59;
- compatible with the test evaluation part, so the output is stored in a way that can be fetched in the evaluation part.

The tool for executing evaluation of test result should be:

- compatible with the test execution part to be able to fetch the actual data;
- able to store the result in a database, preferable the same kind of database that the test cases are stored in;
- able to compare expected outputs with actual outputs, with a comparison algorithm that can be designed by the user;
- able to use different comparisons algorithms for different expected outputs.

The tool for generating the test report should be:

- able to produce a test report according to the IEEE 829-1998 standard [IEEE 1998];
- compatible with the evaluation part so it is possible to fetch the evaluated data;
- able to fetch data from the database that contains the test cases.

9.3.2 Mercury Functional Testing for Wireless

This software supports the automatic testing of cell phone software directly on the real device, which must be running Symbian OS. It comes in two parts: one that is installed on a normal PC and a test

program that is installed on the cell phone where the application is intended to be run. The test software running on the phone sends test data back to the PC over Bluetooth, infra red, USB, and serial connection. This software cannot be used for our test automation project since it only supports Symbian OS.

9.3.3 Automation Anywhere

This is in fact not really a test tool, but it is a tool for automating any task that is performed with a keyboard and a mouse. The reason for looking into this tool was, that it might be used together with existing software, see “9.4.2 B4 test system” on page 70 and “9.4.1 Auster” on page 68, for sending input to the i.box.

The first reason for not choosing this tool was that it would be a long error-prone task to create test cases, because all test cases would have to be recorded by doing them manually. If one would make a mistake when recording the test case, the whole test case would have to be recorded once again from the beginning. Since the test case execution is quite long, it would take much time to record a test case, and because of the length, it would be quite likely that errors would occur during recording.

The other reason, which is the main reason for not choosing this tool in the automation process is that it would be hard if not even impossible to synchronize the use of all tools that would have to be used for simulating all kinds of input.

9.3.4 Other tools

Some other tools for automated testing have also been investigated. These tools include:

- Mercury WinRunner
- Mercury TestDirector
- Compuware TestPartner
- National Instruments’ Labview
- Android

What they all have in common is that they either could not be used with embedded systems, were not compatible with the i.box or that they could only perform GUI testing. Therefore, they were all rejected to be used in our implementation of automated testing.

9.3.5 Result of analysis

Even if several tools on the market offer products which have powerful testing functionalities, did none of the analysed tools fulfilled all the requirements from “9.3.1 Requirement on tool” on page 65. That leads to the decision to develop on application specifically for testing the location functionalities at system level testing in the i.box.

9.4 Design of test execution

One main idea behind the design of the application was that it should be possible to expand the application to not only be used for testing positioning functionalities but also other functionalities in the i.box. That was the reason for dividing the applications different parts according to a module based design. An overview and description of the different modules can be found in “9.4.3 Design overview” on page 71.

Two testing tools, Auster and B4 Test system, where already integrated in the manually test procedure at IVU Traffic Technologies AG. Both of them offered functionalities that were suitable for automated testing and they were at disposal to be used in automated testing. They are further described in “9.4.1 Auster” on page 68 and “9.4.2 B4 test system” on page 70.

9.4.1 Auster

Auster is a tool built at IVU Traffic Technologies AG which can be used for instance to facilitate the test procedure. Since it is built with a wide range of functionalities and can both handle input and output operations it is suitable for being a part of automated testing. Auster is communicating with the software on the i.box via a TCP/IP port and can thereby reach information about the software that is unreachable for a user of the i.box.

9.4.1.1 Button simulation

A functionality that is useable when automating tests is Auster's functionality to simulate button pushes. The signals that comes from a keyboard or a touch screen is simulated with Auster, that means that by using this functionalities one can simulate buttons being used closely to how it would be used by the end user. To get it closer to a real situation when using a simulation one would have to have some kind of robot that an automated test tool can control.

9.4.1.2 Communicating with Auster

It is possible to communicate with Auster via a GUI or a TCP/IP port that is opened by Auster. All the functionalities that are useful to use for testing location functionalities are reachable through the GUI of Auster as well as through sending commands to the TCP/IP port. Auster presents its output by writing all output at the TCP/IP port, in a log file and in Auster's GUI. Writing on the TCP/IP port and at the GUI is performed continuously, while writing to Auster's log file is done approximately every 15th seconds.

9.4.1.3 Output from Auster

To be able to control what information to receive from the Auster, output is divided into different groups that can be set to different output levels (one to seven). For instance, if one is interested to see information about the positioning, one sets the positioning group to a low number, and the other groups to a high number meaning that only the most important events is being shown from all groups except the positioning group, which shows more information. When the filters are set in Auster, Auster communicates with the i.box so the i.box writes the wanted information on the TCP/IP port. Besides getting output by setting levels of the filters it is also possible to invoke functionalities that fetch certain information periodically or only once. To get data that is given by these functionalities it is required that the filter that corresponds to the group that the functionality is within, has been set to a level that allows the stream of information that the function call give as output.

9.4.1.4 Use of Auster in automated testing

The using of Auster can be divided into three parts: to configurate what information is interesting as output, to be able to read the state of the i.box and its status, and to send input signals that represent different button pushes. All usage of Auster was performed by communicating through a TCP/IP port.

To configurate what information that would be presented, filters where set to appropriate levels. One setting was made that periodically fetches information and a command was invoked that presented information once. To be able to view all the wanted information about the actual output from executing the test, the filter belonging to the positioning group was set to highest level, as also the filter belonging to the GPS group. Filter controlling the dialogue events was set to level two, to be able to get information about what was presented at the GUI. In addition to the filter settings the function "pos_info" and the function "loc_trigger_switch" was invoked. The function "pos_info" presents information once about for instance if the vehicle has driven away from the route that it is set to drive and information about the status of the trip, which makes it possible to see if the trip has started or not. The function "loc_trigger_switch" periodically returns information about which was the last net point that the vehicle passed, distance from the last net point and what the next net point on the trip is.

The functionality in Auster that writes output from the i.box to a TCP/IP port was a good way to get feedback from the system. This feedback contains information about the test execution and since the information was available during the execution of the test cases, it was sometimes possible to use this information to control correct execution of the test case.

The possibility to send buttons was made possible by using the function "set_dlg_event", which gave one required button press as an input to the i.box.

9.4.2 B4 test system

B4 test system is a system which is used in testing by controlling inputs to the i.box. The program is used via a GUI and is built up of several DLLs. When performing manual testing the system is mainly

used for sending impulses to simulate the movement of the vehicle and to simulate a door opening or closing. Since the system is based on DLLs it is suitable to be used in other applications, as in an automated testing application.

9.4.2.1 Use of B4 test system in automated testing

The B4 test system was used to be able to send impulses to simulate movement and door openings and closings in an automated test. The movement was executed through setting the speed and the total distance that the vehicle should move in a DLL that communicates with the i.box. To be able to open and close the door, different outputs were sent to different ports on the i.box. The type and configurations of the i.box decided to what ports to send the signal, and what signal that should be sent, i.e. if it was an i.box with an active high entrance or active low that meant that the door was closed.

9.4.3 Design overview

As written earlier, the test execution tool was designed using module based concepts. The following main functionalities that could be divided into different modules was identified as: fetching test cases and test input, controlling the execution of test cases and their actions, sending input signals to the i.box, reading output from the i.box and storing the output for later evaluation.

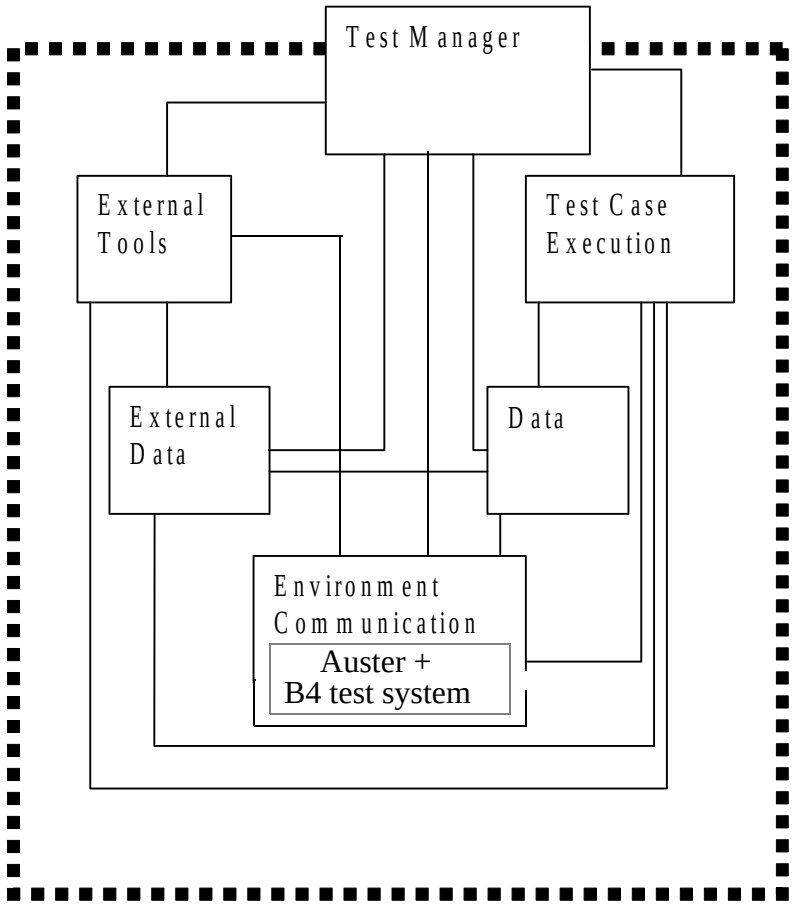


Figure 9.4: Overview of the different modules in the test execution tool

9.4.3.1 Test manager

This holds functionalities that are invoked first during program execution and is at the top layer of the program. Before the execution of test cases can be started, the test environment has to be set up. In the case of this tool, the set up that has to be done before running a test suite is to start Auster and clear old data in the database that is used to store result between the test performance part and test evaluation

part. After the set up procedure the test manager sends a command to external data manager to fetch the test cases that should be executed. As a last part the test execution module is called to iterate over the test suite and execute each test case. The flow of events that occur in this module can be seen in ”Figure 9.5: Flow of test manager.” on page 73.

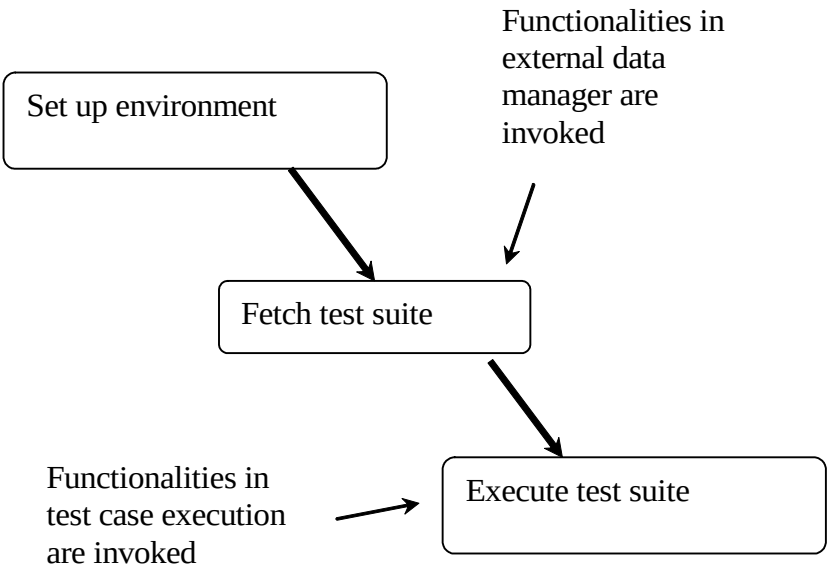


Figure 9.5: Flow of test manager.

9.4.3.2 External data manager

The test cases are stored in an Oracle database, which the external data manager is responsible for handling. When each test case is run it can need additional information to calculate the input for each action. This information can be things like: which stops are in a specific route that is used in the test case, how far is it between each stop in the route and what kind of hardware set up does the i.box under test have, e.g. which port are the doors connected to and does a high or low signal on the port mean that the door is closed. This information

is either read from a database if it is information about routes or from a file if it is information about the i.box under test.

After the test is run, the output has to be stored somewhere for later use during the evaluation of the test cases. The same database which stores the test cases is used for storing the outputs. If something goes wrong during the test execution, either if something is corrupt in the test case or an execution error in the test performance tool, information about the error is stored in the database to make it easier for the user to discover what went wrong.

9.4.3.3 Data manager

In order to not have to fetch the same information from a database and from a file several times during the execution of a test, a structure for storing data in memory during test execution was designed. This was necessary both because it is faster accessing data from memory than executing database queries and accessing files, and because the different modules need persistent data storage and the need to share information between different threads used in the application.

9.4.3.4 Test case execution

All the functionalities that are bounded to execution of a test case are gathered in this module. The first thing that occurs in execution of a test case is an initialisation part, that makes sure that the application, the environment around the i.box and the i.box are started at initial state in all the executions of test cases. Parts in the initialization process are for instance to connect Auster to the i.box and to configure what outputs that should be logged during execution. When all the set ups have been performed, the focus in the application is moved from initialisation part to execution of the actions in the test case. Depending on what actions that are stored in the internal storage for each test case, the intended action execution process will be invoked. The focus in the program is moved to handling storing actual data after the execution of the last action in the test case is finished. When the storing of the actual data is completed, the execution of next unexecuted test case will be started. The flow that takes place can be seen in ”Figure 9.6: Flow of the test case execution.” on page 75.

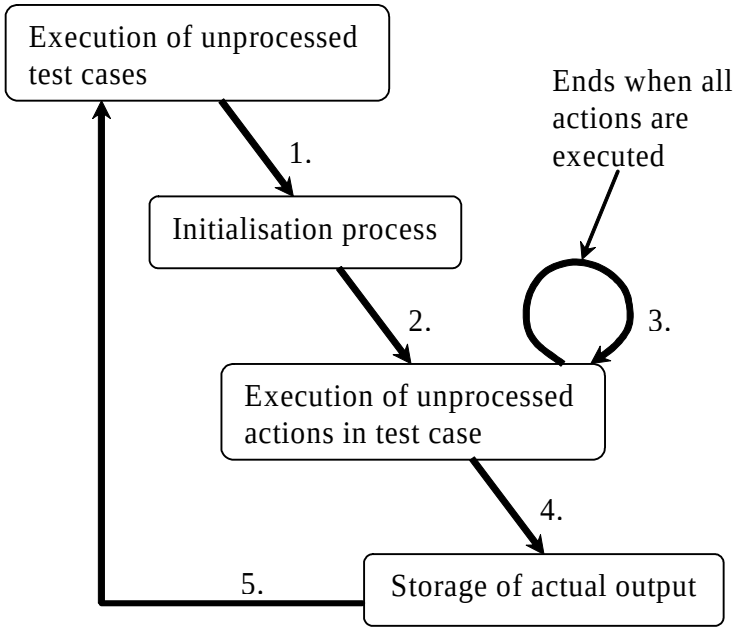


Figure 9.6: Flow of the test case execution.

9.4.3.5 Environment communication

The functionalities that communicate with external applications or the i.box are gathered in this module. As shown in the picture, ”Figure 9.4: Overview of the different modules in the test execution tool” on page 72, the external components that are used are Auster and B4 test system. The direct communication with the i.box, that is used to send GPS coordinates, also belongs to this group as well as the telnet client that is used to reboot the i.box.

9.4.3.6 External tools

Three large functionalities (logging, conversions and timer) that are required at several places in the program can be found in the external tools module. In common for the three parts is that each of them can be considered to be independent from the rest of the program.

The logging part is the part that is mostly connected to the other parts of the application. In general it is used for storing outputs from the i.box and to write error in a log file. That means that it handles: storage of actual data, logs different outputs from the i.box that are used during program execution and error handling when it is not possible to store the error in the database.

The conversion part handles all the necessary conversions in the program. The most advanced conversions are conversions between GPS coordinates in formats as DMS, DM, DD and CC.

The timer contains functionalities that could be found in a stop watch: it facilitates possibilities to start, stop, restart and set a maximal amount of time that can elapse and check if there is time left.

9.4.4 Design patterns

Within the creational design patterns Singleton has been used, whenever needed. It is needed to assure that there exists only one object of each of the following: internal storage, output log, GPS sender, connection to Auster and connections to data bases.

Behavioural patterns as iterator has been used to iterate over, and get information about test cases, actions, expected values and net points.

For descriptions of the design patterns see [Gamma 1995].

9.4.5 Concepts

The concepts in this section are concepts that are specifically interesting to look more into. Some of them are issues that are applicable only for embedded systems, some of them regard only testing in real-time system, some of them are interesting only regarding testing the i.box and some are of interest when using automated testing in general. Although most of them are applicable for testing in several of the mentioned systems.

9.4.5.1 When to read the output?

There are some aspects that need to be considered when deciding on the time to read the outputs. Especially when the testing is performed on a system that requires a pre-defined maximal execution time, as

often in embedded systems. The core behind the issue is that one wants to be sure that the value that is read as the actual value really is the actual value and not an old value for a response in the system. One needs to be sure that all the actions that have been executed in the test application also have been executed in the object that is tested. The best solution would be to read the output exactly from the time point when the last action in the test case has been executed in the test object. Unfortunately this situation is hard, and perhaps even impossible, to achieve in many test executions. The underlying reasons that make it impossible are different depending on the test object. When testing the i.box the reasons depends on i.box specified characteristics, that it is a real-time system and that it is an embedded system.

When testing a real-time system, the time to fetch output could be decided based on a defined response time, that would assure that action that has been executed in the test application also have been executed in the test object. In the case of testing the i.box, there does not exist a well specified response time for executing each of the actions, which means that it is not possible to be totally certain when the test object is finished with the execution.

There will also be a delay from the time point when the execution in the i.box is finished until the output has reached the test application. The reason for that is that the output is only possible to be read through the Auster and not directly from the i.box. To be able to minimize the delay one would have to read the outputs directly from the software at the test object, but since it is an embedded system the hardware is not meant for having a test application running; it is designed for the software of the main application. The next question that perhaps then comes to ones mind is: why not move the software out of the environment and test it in an environment that it is suitable? But that is unfortunately not that easy to perform, since then it would not be a system test. One wants to test the behaviour of the application in the environment that it should be run on when the application is in use.

Another issue that one needs to consider when designing an automated test application is that the output value could change from a correct value to an incorrect value just after the value is read. It can therefore be of interest to not only read a single value, rather to also

investigate the values over a period of time. All the scenarios that are viewed in ”Figure 9.7: Example of different output situations.” on page 79 would be possible to get as actual values. If the output would be read at different time points, different actual values could be fetched, which could lead to false results. The execution of the test case could be seen as a success, even if it contained errors.

In this implementation, actual values are read during a period of time, to be able to fetch any output values that may differ. The time point when to start reading the outputs as actual values is when the last action has been executed in the test application plus response time that is specified for each different i.box system. As mentioned earlier there does not exist a well defined response time for all the different actions that should be executed, the response time that is used here is a response time that is a time that is the same regardless what events that is performed in the i.box. The response time that is defined must therefore be the longest time that any action requires. That means that the system will be waiting too long after some execution and the situation described in ”Figure 9.7: Example of different output situations.” on page 79 could unfortunately be missed in cases, when the execution requires shorter time than the specified response time. The time point for when to stop reading actual values is calculated so the duration of the fetching period always is ten second, which is a value that have been selected based on testing.

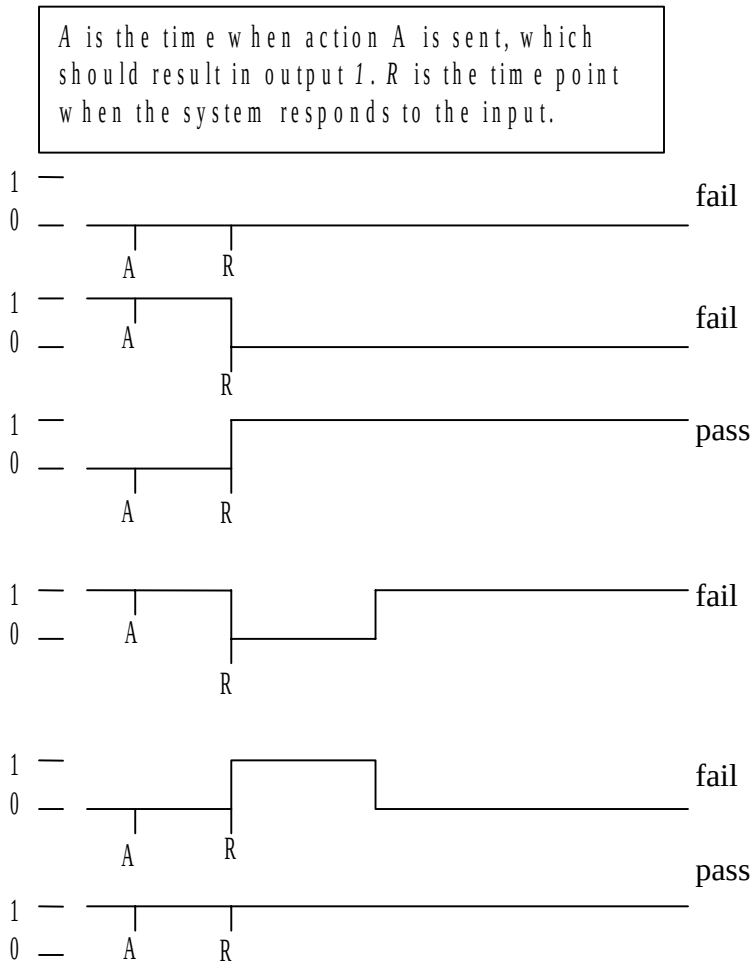


Figure 9.7: Example of different output situations.

9.4.5.2 How to assure that the test cases always are started with same system preconditions?

To be able to run automated test cases the system needs to be in the same state before the execution of each test case, otherwise it would not be possible to predict what events will be triggered for the executed actions, when designing test cases. At the end of execution of an test case the i.box is in different states depending on what actions

have preceded and what i.box system that is tested. That means that something needs to be performed to set the i.box in the same start state before the new test case can be executed. One solution would be to have an ending procedure that could be executed after the actual value is read, which would set the i.box in the correct start state. But since the i.box is in different states depending on which actions are in the test case that is run, what i.box system is tested, and that the same test cases are used for all different systems, this is not possible in this study. Also, if for some reason the test case execution fails, either because of malfunction of the i.box or that some input signals can get lost on its way to the i.box, the i.box would be in a completely unknown state. Which would set the i.box in one of a large number of different states at the ending of a test case. This is also the reason for not being able to have dependent test cases, that is, the next test case is dependable of the outcome of the last test case and can continue to run directly where the last ended, since the state of the i.box is very different from test case to test case.

Another solution is to reboot the i.box system after every test case execution. That results in that the i.box is always in the same start state before running a new test case, independent of the result of the previous execution and what actions that has been executed in the i.box. It will affect the execution of the tests in the way that every execution requires more time. The time it requires to reboot is approximately two minutes. This last solution is the way it is implemented in this study.

9.4.5.3 How to handle execution errors?

It is important to store as much information as possible about the errors that may occur during a program execution, since in most cases these errors also cause errors in the result of the execution of the test. For instance if the test application runs the actions in an order that differs from the order that is specified in the test case, the expected value and the actual value would most likely differ and an test case failure would happen. In that case, a test case failure is created even if the actual source of the error is perhaps not in the test object that is tested, but in the test application. This is the reason for error handling being of extra high importance when performing automated testing. In this study, the errors are stored either in the database for actual

values, or in an error log file. The first choose on where to store the error should always be the database, which makes it possible to include the error in the test report. If the errors occur when it is not possible to write in the database, the error should be written into the error log file.

9.4.5.4 How to erase differences between different test objects?

To be able to construct an automated test application that can run one set of test cases on different types of test objects, it is needed that all the dissimilarities that exist between test objects are erased. Since changing the test object is not a possible alternative, the test application need to equalize the objects. That can only be done if the discrepancies are defined and the test application have access to the information. A solution, which is implemented in this study, is to document the differences in a configuration file which the test application can read from. When one is performing an event that is different in different i.box system, it should be defined in the test case that the configuration file should be used.

The dissimilarities between different i.box systems that affect system testing of the positioning functionalities are: sequences of button pushes may result in different states, start procedure is performed based on different types of information and different configurations of pins on the connector that is attached to the i.box.

9.4.6 Implementation notes to the test execution

The implementation was done using C++ in Visual Studio. Class diagram that describes the implementation can be found in “Appendix A Design diagrams” on page 101. Four threads are used during program execution. One thread continuously logs outputs on a TCP port, one triggers the i.box to dump specified output information, one sends GPS coordinates and also the main thread that executes the program.

9.4.7 Distance calculations

There are a number of different formats in which a GPS coordinate usually are specified: DMS, DM and DD. All of them are used during the testing of the i.box. The coordinates that are saved in the data records for the lines are stored in the DMS format, the GPS coordinates that are sent to the i.box in the GPRMC and GPGGA messages are in the DM format and the i.box uses the DD format when making calculations with coordinates. They all have in common that it is not trivial to calculate for example all GPS coordinates on the straight line between two given coordinates, because the coordinate system used in these formats are not linear, since they are made to be used for describing locations on the surface of the earth. The DD format is a decimal system with the base 10, so it can be used for rough calculations on small ranges. How large the errors of the calculations are depends how far from the equator the coordinates in the calculations are, the farer from the equator, the greater the errors will be. To be able to use simple mathematical formulas in the above mentioned calculation that have minimal errors, the coordinates have to be transformed to a linear coordinate system. For this purpose, there exist a standard coordinate that is widely used for map projections: UTM (Universal Transverse Mercator coordinate system). It is based on the transverse mercator system.

9.4.7.1 Transverse mercator system

The projection is done by using a cylinder that touches the surface of the earth. What is seen straight through the cylinder along its normal is drawn on the cylinder. By using this method, the earth is correctly reproduced at the centre of the cylinder but will be wrongly reproduced at the edges of the cylinder. When the earth is drawn on the cylinder, the cylinder are cut and unfolded to a plane which will leave us with a oblong map. The two steps of drawing the map on the cylinder and cutting the cylinder are illustrated in ”Figure 9.8: The cylinder touching the surface of the earth.” on page 83 and ”Figure 9.9: The cut up cylinder with the earth drawn on it.” on page 83.

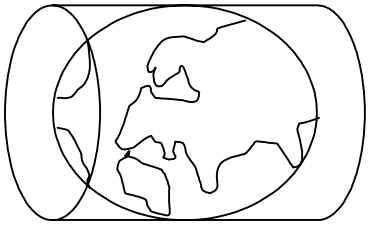


Figure 9.8: The cylinder touching the surface of the earth.

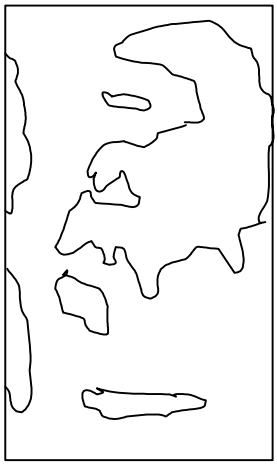


Figure 9.9: The cut up cylinder with the earth drawn on it.

9.4.7.2 Universal Transverse Mercator coordinate system (UTM)

UTM divides the earth in 60 zones which together cover the whole surface on shore of the earth. A zone is 6° wide and stretches from 80°S to 84°N. Zone 1 starts at 180°W and ends at 174°W. Each zone is in its turn divided into 20 latitude zones. They are named after the

capital letters in the english alphabet starting at “A” and ending at “X”. The letters “I” and “O” are omitted because of their resemblance with the figures “1” and “0”. All latitude zones stretch over 8° in the south-north direction, except zone X which stretches over 12° to cover the most northern land of the earth. Therefore, zone A has its southern border at 80°S and the northern border for zone X is at 84°N.

By mapping each zone separately with an adopted Transverse Mercator Projection, the distortion can be limited all over the surface of the earth. By setting a scaling factor of 0.996 at the transformation, the distortion will be evener over the zone. The distortion will with this scaling factor be zero along two south-northern running lines approximately 180 kilometers from the zone median. A measured distance will be 0.4% too short in comparison to the reality along the zone median and 0.1% too long by the zone border.

A latitude zone is denoted by “zone number” “latitude character”. For example, Sweden is covered 32V, 33U, 33V, 33W, 34V and 34W.

The zones are homogeneous distributed over the globe with two exceptions: The border between zone 31V and zone 32V is moved to the west to cover the small land that otherwise would be covered by zone 31V. The other part of the earth which not has homogeneously distributed zones is the area around Svalbard. The zones 31X, 33X, 35X and 37X are widened to cover the areas that the zones 32X, 34X, and 36X otherwise would have covered. These last three zones are not used.

A position on the earth is in UTM described as longitude zone together with a northern/eastern coordinate pair. The northern coordinate is the projected distance from the equator to the point (positive values on the northern hemisphere and negative on the southern hemisphere) and the eastern coordinate is the projected distance from the zone median (positive values at the eastern side of the median and negative values on the western side). To avoid negative numbers, the median is given a eastern coordinate value of 500,000, which means that the projected distance is increased by this value for all eastern coordinates. To avoid negative numbers for the southern hemisphere, the northern coordinate is increased by 10,000,000, but

only on the southern hemisphere, and not everywhere as for the eastern coordinate.

9.4.7.3 CC

The CC format is an extension of the UTM, that is a mapping to a cartesian coordinate system that is applicable around the globe. This system uses a global coordinate system so that calculating distances, finding coordinates along a line, and other coordinate calculations are simplified. This system is not a universal standard, but instead a specialisation for this implementation made by the authors of this thesis. This format is used since it is easier to calculate a line between two points that in the UTM belong to a different zones, than it is with the UTM format.

9.5 Design test evaluation

The test evaluation part in this automated test procedure contains functionalities like evaluation of the result and creating a test report. The information that is needed is fetched from the test case database and where the actual values are stored. An overview of the evaluation tool is shown in ”Figure 9.10: Overview of test evaluation tool” on page 86. As can be seen in the figure, the test report is created within this tool. More information about the test report can be found in section “9.6 Generate result” on page 89 and in “6.5 Test result” on page 39.

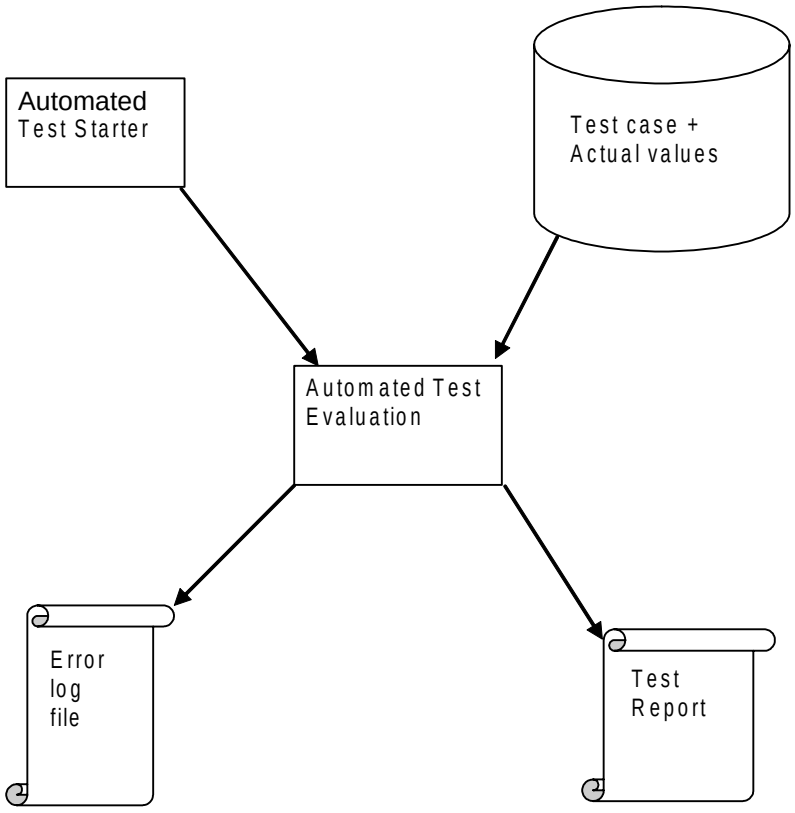


Figure 9.10: Overview of test evaluation tool

9.5.1 Evaluation

The result of the evaluation is passed or failed. All parameter values that belong to an expected value in the test case need to have pass-results, in order for that expected value to be considered to be passed. And all the expected values in a test case need to have pass-results for the test case to be passed. This procedure is described in ”Figure 9.11: Example of an evaluation with pass result.” on page 87 and in ”Figure 9.12: Example of an evaluation with fail result.” on page 88.

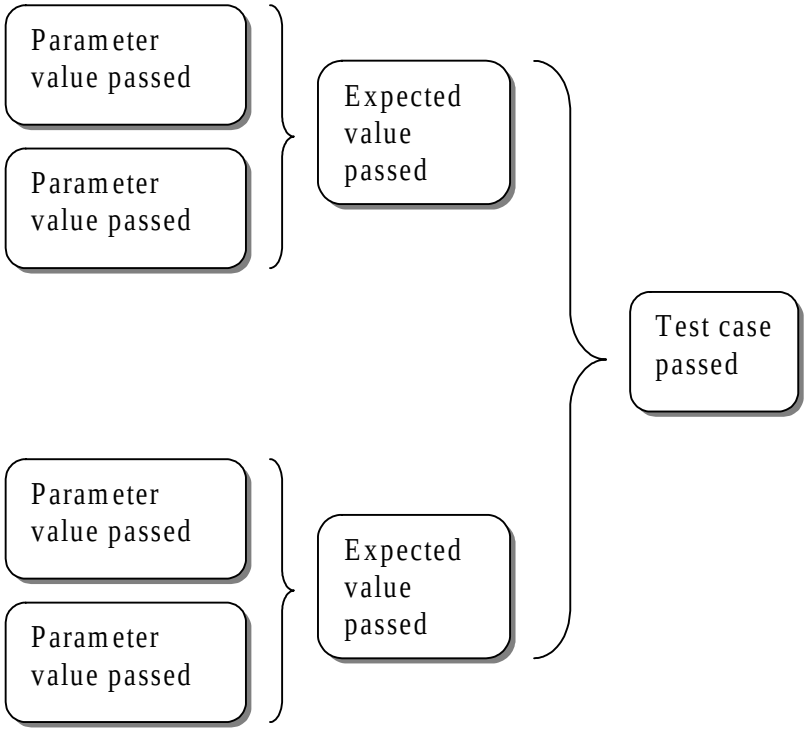


Figure 9.11: Example of an evaluation with pass result.

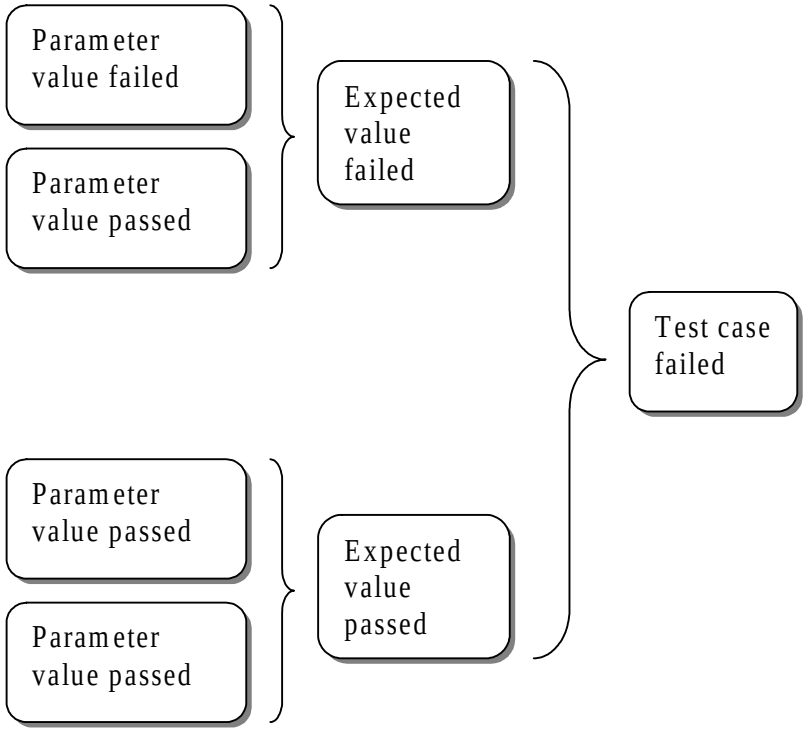


Figure 9.12: Example of an evaluation with fail result.

As written in “9.2.4 Expected output” on page 63, some expected values have parameter values that cannot be handled only with an equality comparison. There exist two parameters with conditions that affect the evaluation procedure. All actual values that respond to the parameter value “Distance since last net point” in expected value “Position between net points” that are larger than zero are valid actual outputs. If the value is zero, the expected value will fail no matter the other two parameters’ values. The expected value “i.box has been synchronized” is the other expected value with special condition. If the parameter “Distance since last synchronisation” has value “synchronized”, only the actual value zero is a successful output value. If the parameter has the expected value “not synchronized”, more than one value should result in success. Namely, all the values that

are larger than zero means that the i.box has not been synchronized, and should result in pass in the evaluation procedure.

Since the actual values are saved during a period of time, it is possible to make evaluation based on several values. The reason for wanting to do so is discussed in “9.4.5.1 When to read the output?” on page 76. As viewed in “9.7 Example of different output situations.” on page 79 the only situation that should result in a passed result is when the actual value do not fluctuate during the time period. All result are saved in the test report when there exist a fluctuation, otherwise only one result is written in the test report.

9.6 Generate result

As written in chapter “9.5 Design test evaluation” on page 85 the creation of the test report was integrated in the second part, the evaluation part.

9.6.1 Format

The test report was made according to the IEEE Std 829-1998 standard [IEEE 1998] with some modifications. The two documents, test log and test incident report, where concatenated into the test report. Information about who has run the test case are omitted, even though it exists in the IEEE standard. The main parts of the report consists of information about the test execution, the test case, its preconditions, actions, expected result and the actual result.

9.6.2 XML file

The test report was chosen to be stored in an XML file, so it is possible to view the result in both a GUI and with help of an XSL file in a web browser. The possibility to view the test rapport in a web browser makes it easier to share the result between customers, developers, QE, project managers and other people involved.

The name of the file is ATE_result added to a time stamp. The time stamp is added to the file name to facilitate administration of test case executions.

9.6.3 XSL presentation

The XSL file that was built views a summary of the test case execution with a picture describing the result. In the summing up part any errors that may have occurred during execution of a test case are also showed. Further down in the file is all information corresponding to each test case showed. An example of a test report visualized by using XSL is shown in ”Figure 9.13: Example of a test report as it can be presented in a web browser formatted with an XSL-style sheet.” on page 91.

Result from Automated Test Execution

Result

Test Case ID	1
Result	passed

Test Case Details

Test Case

Id	1	bereich	Bram
Name	Drive to last stop	basis_version	1.0
Description	Drive to last stop and check info about next stop	linie	725
Author	Hubert	fahrt	17
Version	1	kurs	03
Last Update	31.05.07 16:05:11	umlauf	200003
Creation Time	31.05.07 16:05:11	gps_status	1
		wi_status	1
		time	2007-05-14 14:24:30

Actions

1. Push button buttons : confirm_trip.
2. Wait wait_time : 5.
3. Push button buttons : ok.
4. Set position gps_x : 43306549. gps_y : 515116909.
5. Drive to position stop_point : 53280290.

Output

Next net point	Next net point	Expected value: -1 Actual value: -1	passed
Position is at net point	Distance since last net point	Expected value: 0 Actual value: 0	passed
	Last net point	Expected value: brnds Actual value: brdns	passed

passed

Figure 9.13: Example of a test report as it can be presented in a web browser formatted with an XSL-style sheet.

PART IV

ANALYSIS AND FUTURE WORK

This ending part of the master thesis presents an analyse of the work that has been done by transforming the manual test procedure into an automated test procedure.

The work of this thesis is not comprehensive within the large area of how a manually testing procedure in a real-time embedded system can be automated; there remain many interesting fields to explore. Some of the fields that deserve extra attention are described in this part.

CHAPTER 10

ANALYSIS

The analysis that is found in this chapter is an analysis of the result of the master thesis.

10.1 Result

The objective of this master thesis was that it should automate a manual testing procedure in an embedded real-time system, which has been attained. The master thesis has shown that it is possible to automate the testing process, by implementing a suggestion that is ready for being used in an automated testing procedure. However many things have to be taken into account when automating testing in an embedded system. For example, it could require special test hardware and software, which was the case for this implementation. If this hardware and software were not already implemented it would have been much harder and would have costed a lot more. Thanks to that these tools were already used for the manual testing, the possibility for being able to implement automated testing increased a lot.

What can be seen at this point is that the implementation of the tool was succesful, but what about the implementation of the automated test process? When this analysis is written, the new automated test tool has only been used for a couple of weeks, and the testers are still exploring its facilities and supposed advantages. To be able to evaluate if the implementation of the new test process was successful or not, the tool has to be used for some months in order for the testing to start benefit from it. This evaluation could be a part of the future work and is shortly described in “11.1 Measure the automated testing” on page 97.

CHAPTER 11

FUTURE WORK

The work of this master thesis is a start for introducing automated testing in a software development process and the work leaves unanswered questions to be solved. These questions are presented in this part. When writing this ending chapter, it has been decided that the project in which this master thesis was the first step will continue. In what way it should continue is not decided, there exist two alternatives in how the work in the nearest future will proceed. These are presented under “11.5 Test of another real-time embedded system” on page 99 and “11.6 Extending test focus” on page 99.

11.1 Measure the automated testing

At the moment of writing this master thesis, it is not possible to measure the automated testing, since the process have just been automated and the measurement of the automated testing should be done during months [Fewster 1999]. Even if it can be hard to measure automated testing, is it not hard to find motivations why one should do it. It must for instance be done to be able to certify if it was a good investment to automate the process, which is useful knowledge when deciding about future progress. Another reason for measuring is to be able to see what changes should be done in the test automation as it is implemented in the current version and to have a solid base if considering other choices [Fewster 1999].

The foundation of what to measure should preferable derive from Bill Hetzel's definition of what a useful measure is: “as one that supports effective analysis and decision making and that can be obtained relatively easily” [Hetzel 1993].

It would be interesting to measure how the result of automated testing responds to the objectives. A suggestion of a procedure to measure it is to, after the system has been in use for a few months, perform a survey to investigate opinions of the user of the automated test. The survey could focus on to what degree the automated test fulfils the following statements:

- runs tests unattended;
- finds regression bugs;
- results in better software quality;
- tests more thoroughly compared to manually testing;
- better use of resources;
- tests more time efficient compared to manually testing.

11.2 Dynamic comparison

Another area that is left to the future is to explore what the result of the automated test would be if one choose to implement dynamic comparison instead of post-execution comparison. The question at issue could be to investigate if the result of implementing the dynamic comparison would lead to faster execution of tests that fail to be evaluated correctly, since the test can continue for several minutes when dynamic comparison could have found that error immediatelly.

11.3 Subscripted tests

According to Mark Fewster and Dorothy Graham [Fewster 1999], one common problem with automated testing is the maintenance of the automated test. The effort it takes to maintain the automated test should not be larger than performing the test manually. It could therefore be interesting to investigate the affect of the possibility to have one test case that executes actions that already are stated in other test cases. Perhaps will that result in a system with fewer test cases that are easier to maintain, or it could also result in finding out that the effort required to introduce subscripted testing is larger than it

takes to have test cases with partly duplicated contents. This question of issue has been left out in the work with this master thesis.

11.4 Test evaluation

There exist more questions within the evaluation part that can be solved. No work has been done in investigating if it is preferable to have some kind of semantic comparison algorithm, which perhaps could be very efficient if the quantity of the expected values would be expanding.

11.5 Test of another real-time embedded system

One possible continuation when this master thesis ends is to apply the methods that have been used during this master thesis and the knowledge that have been achieved during the work, on another system. By automating another testing process on another real-time embedded system general conclusions can be drawn and improvements that are discovered when measuring this automated process can be implemented and evaluated.

11.6 Extending test focus

The area within the test object that has been in focus in this master thesis has been the positioning functionalities within the embedded real-time system, the i.box. A next step can be to investigate what other functionalities in the i.box that could be suitable for automated testing and to automate test processes where it is suitable.

APPENDIX A

DESIGN DIAGRAMS

In this appendix, diagrams are presented that show how an automated test tool that can test the i.box product has been implemented.

A.1 The test performance

Some classes were implemented by third parties. Those classes are: Database, TestCaseDB, TripDB, ExternalDataManager, ConfigFileReader, Timer, Logger, LoggerBlockList, LoggerBlock, LoggerBlockParam, Thread, ThreadException and IRunnable. To get the complete picture of the implementation, these are presented anyway. All other classes are the work of this final thesis.

A.1.1 Internal module collaboration

The following figures shows the internal collaboration for the modules in the project.



Figure A.1: The test manager module.

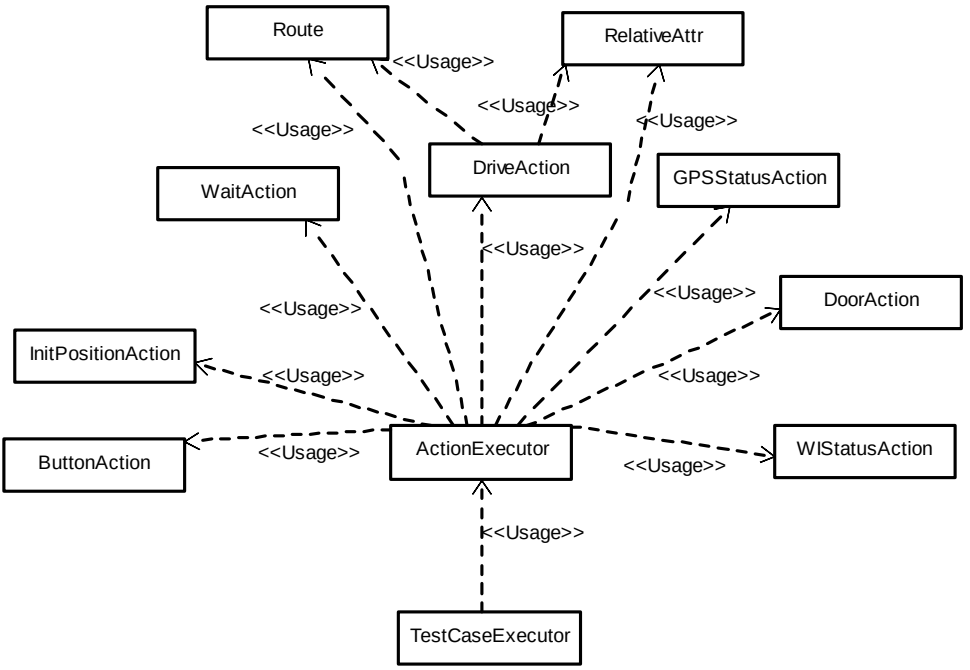


Figure A.2: The test case execution module.

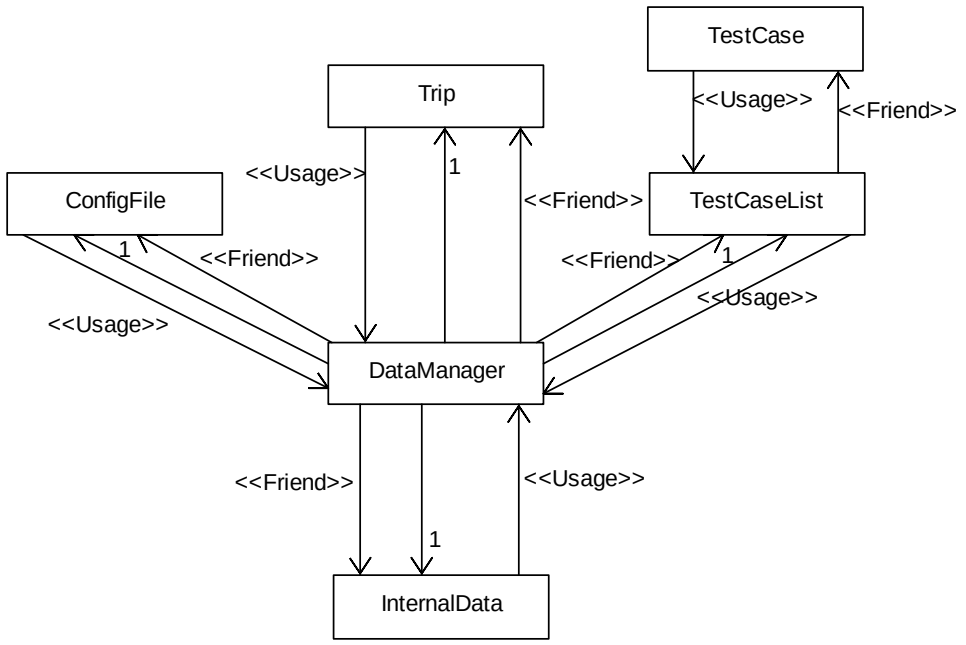


Figure A.3: The data module.

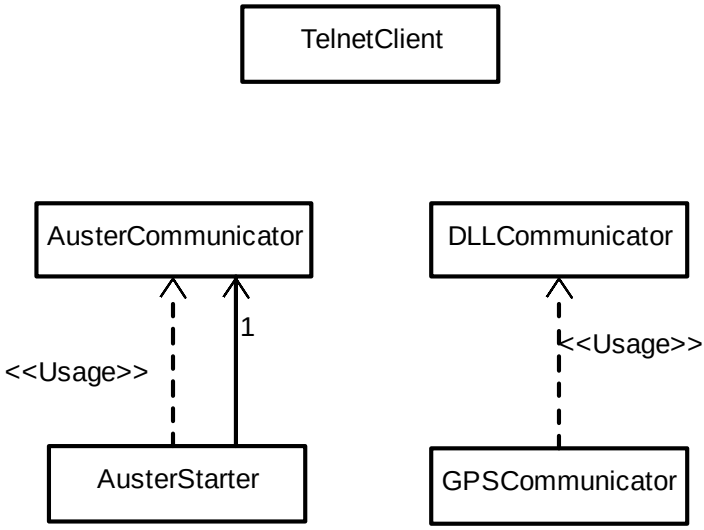


Figure A.4: The environment communication module.

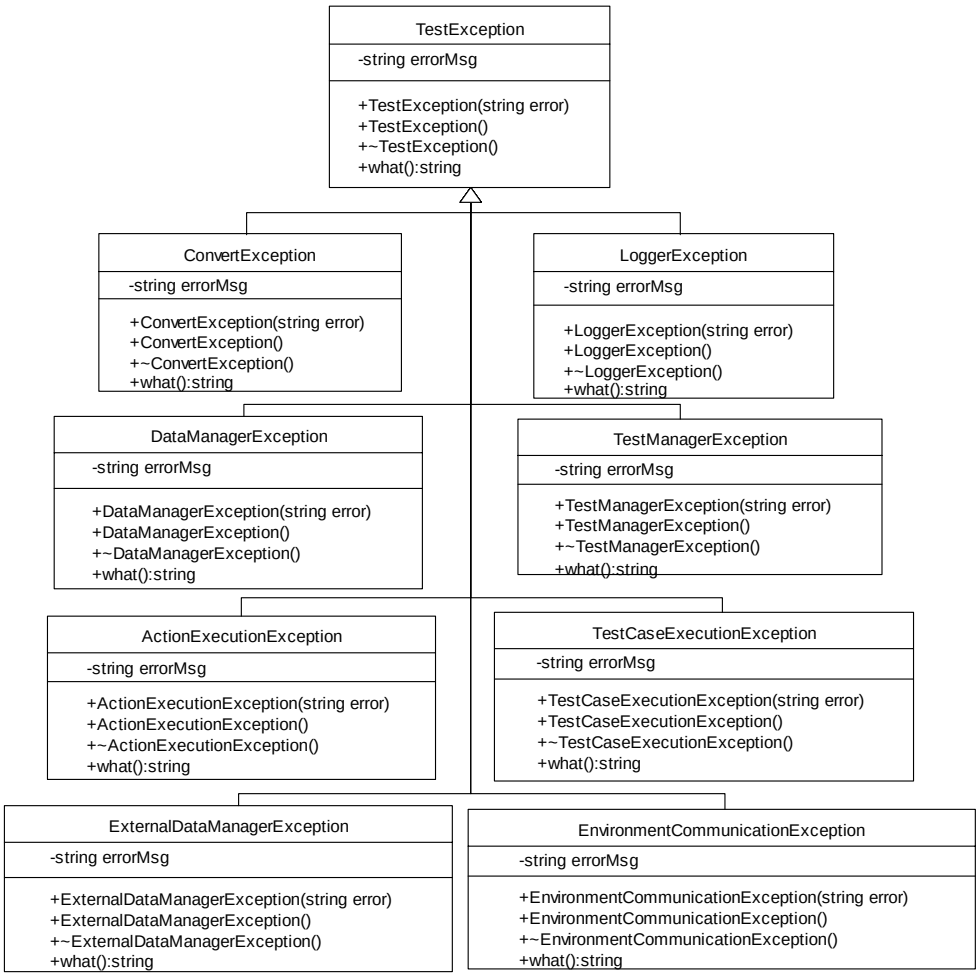


Figure A.5: The exception module

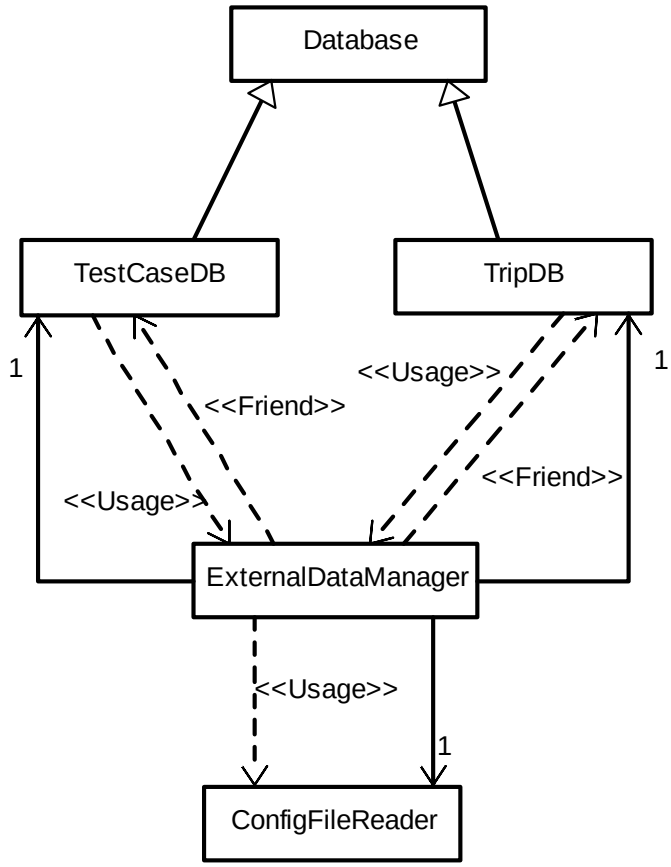


Figure A.6: The external data module.

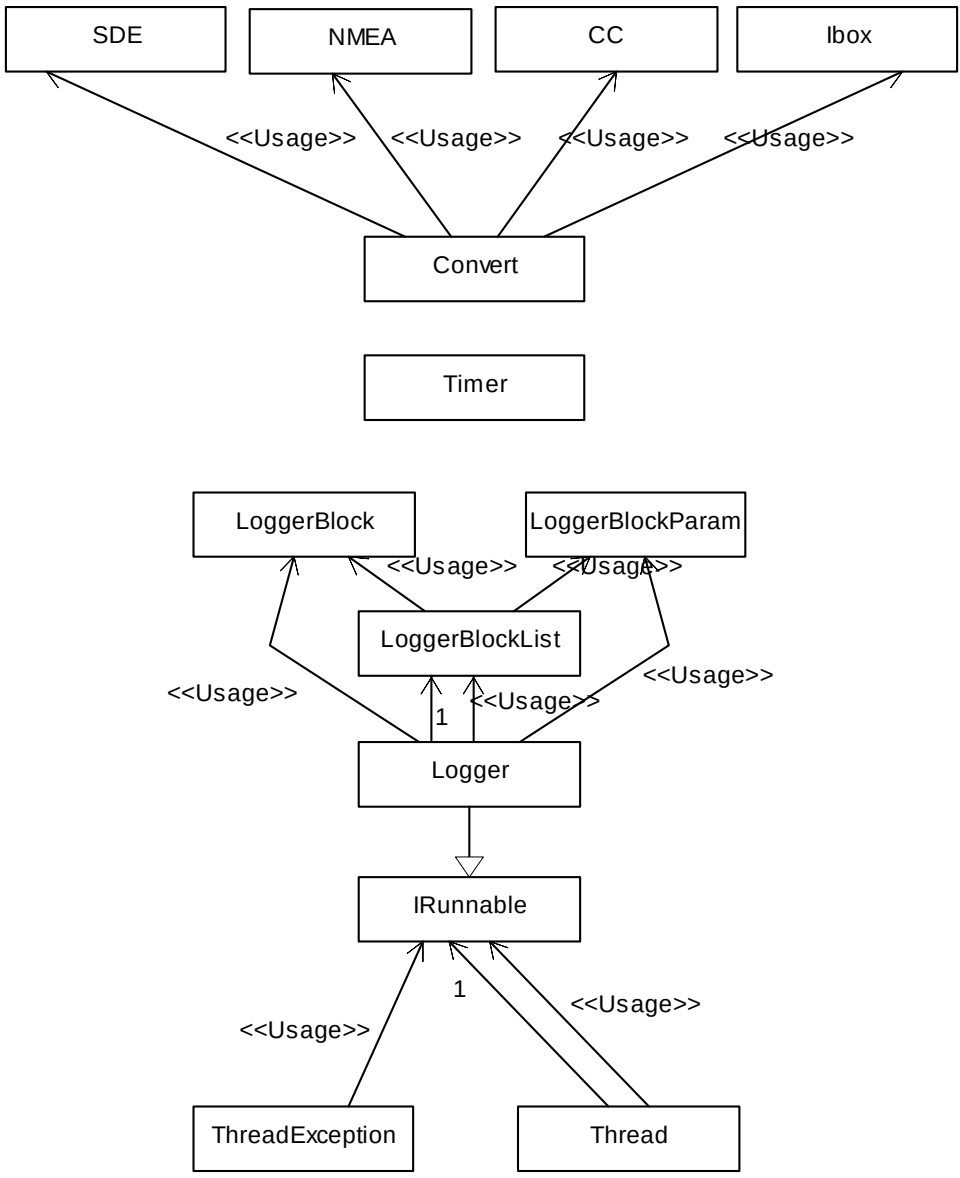


Figure A.7: The external tools module.

A.1.2 Detailed class diagrams

The following figures show the detailed description of the classes including their collaboration with other classes.

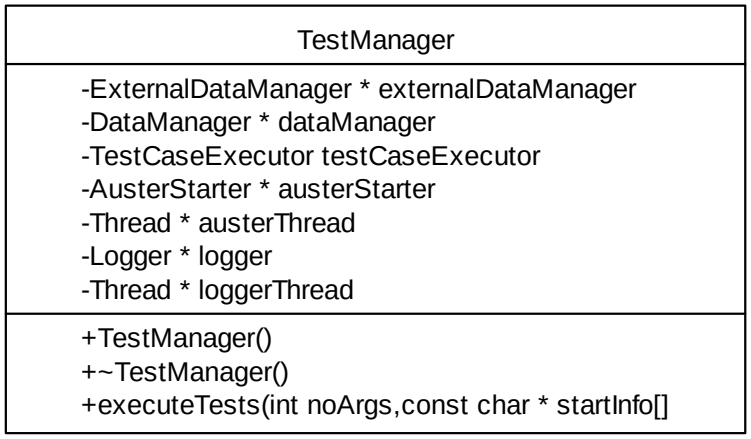


Figure A.8: The test manager class.

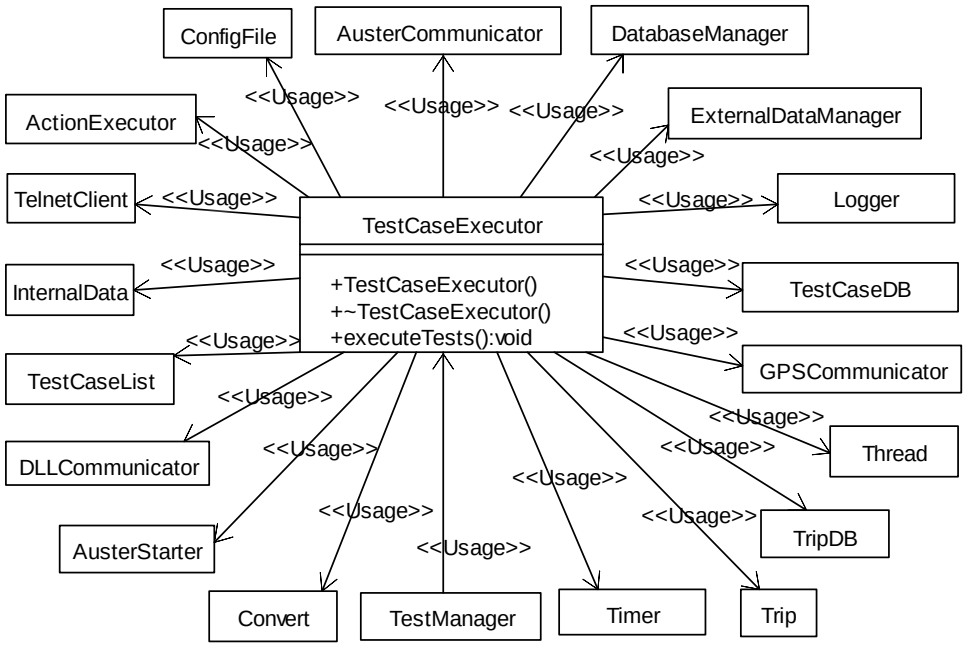


Figure A.9: The test executor class.

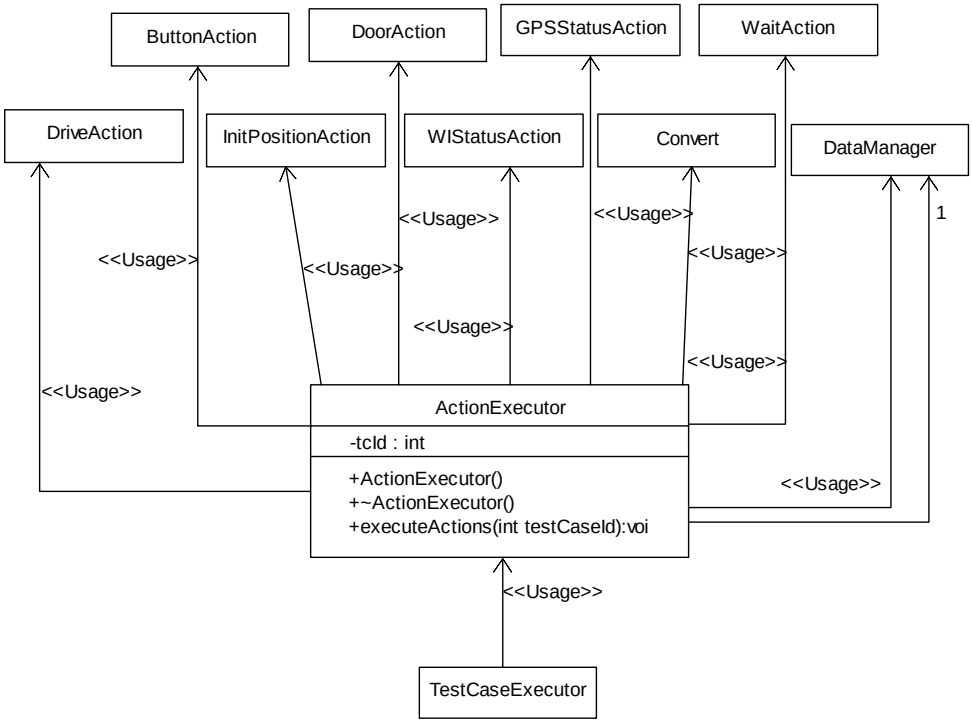


Figure A.10: The action executor class.

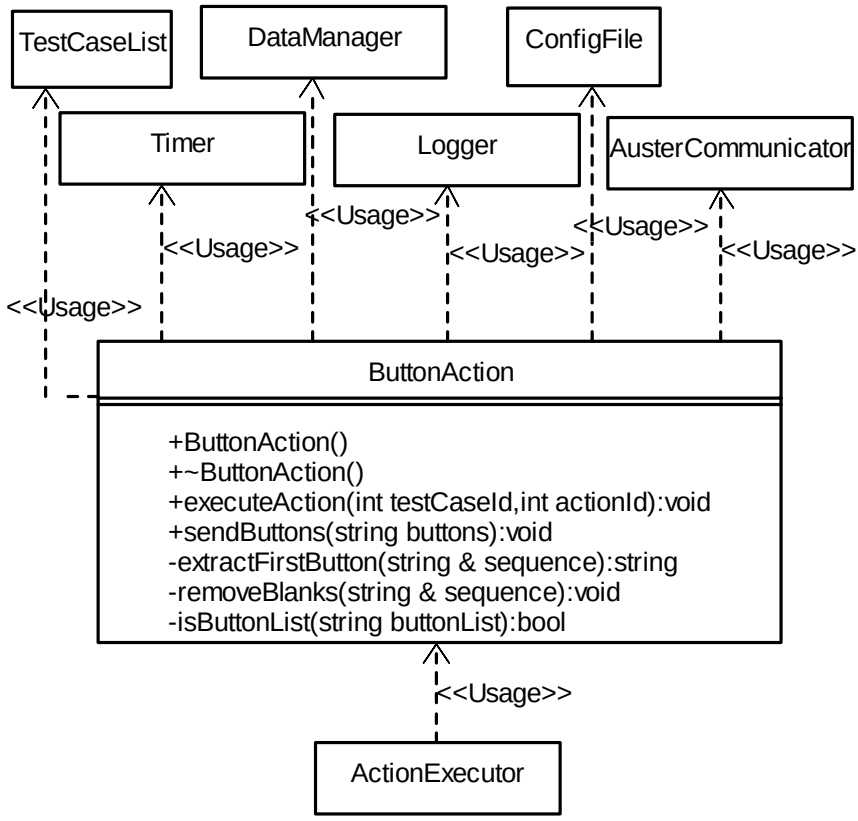


Figure A.11: The button action class.

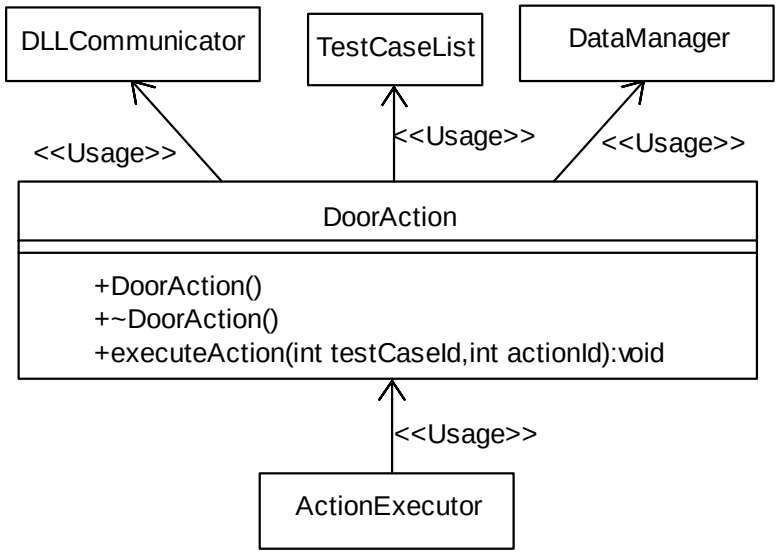


Figure A.12: The door action class.

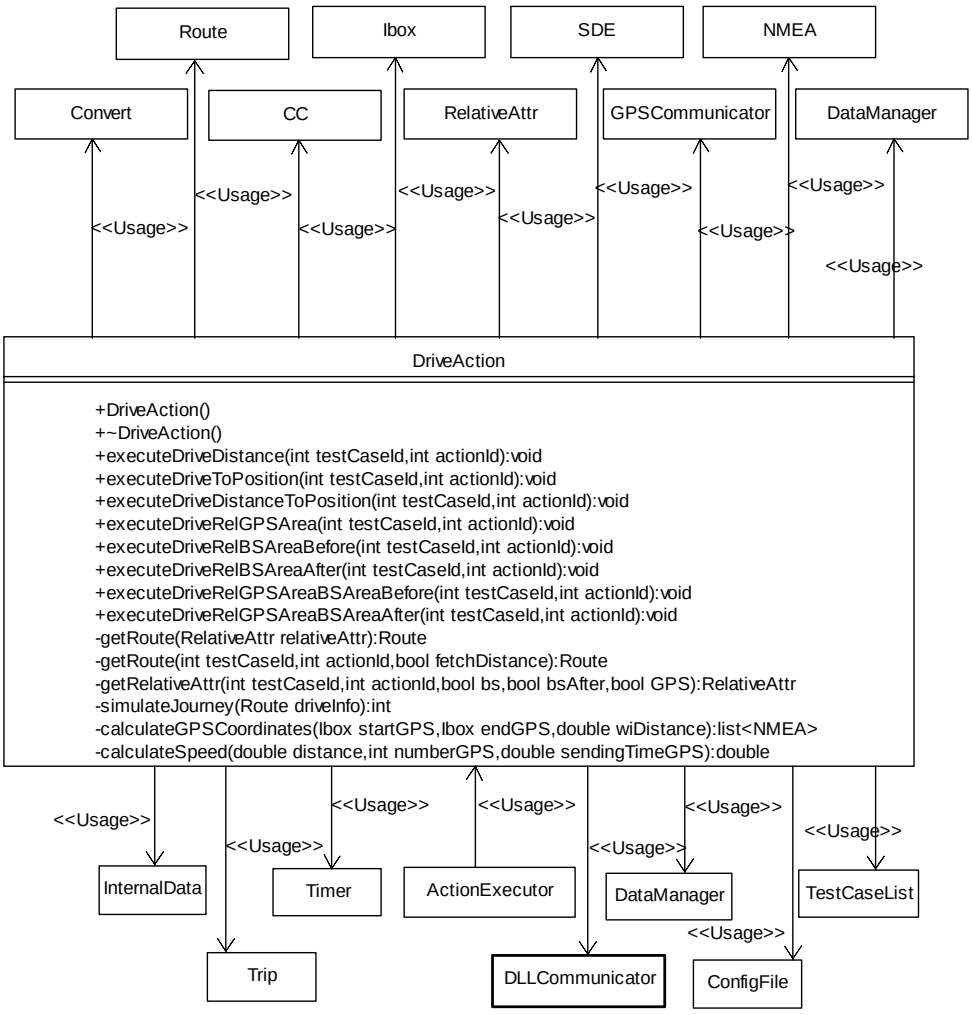


Figure A.13: The drive action class.

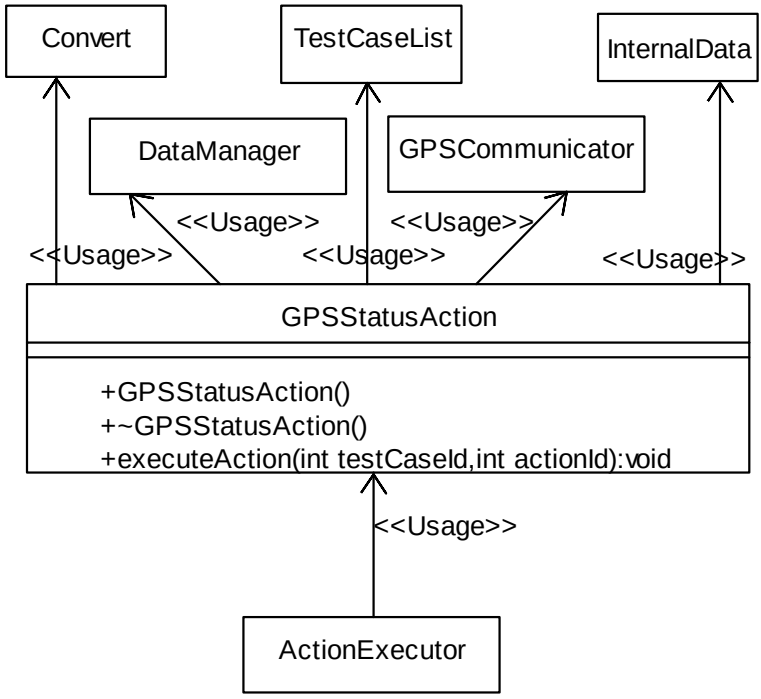


Figure A.14: The GPS status action class.

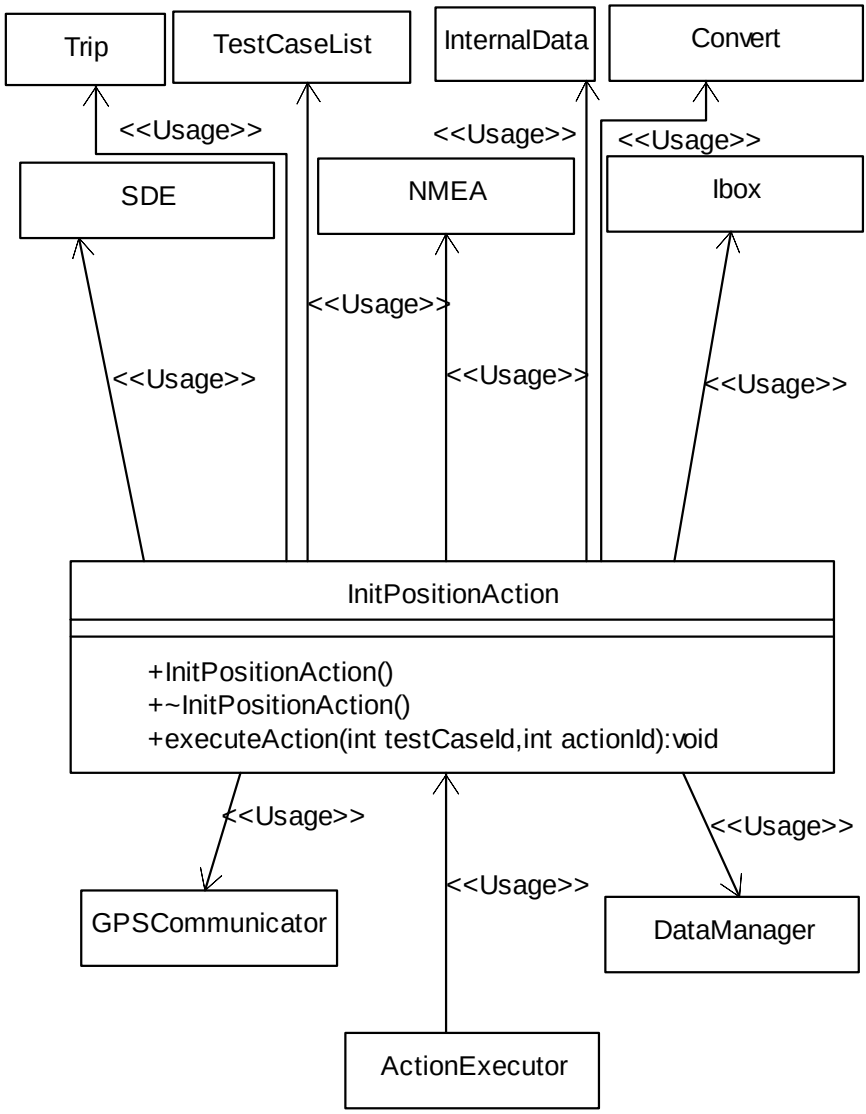


Figure A.15: The init position class.

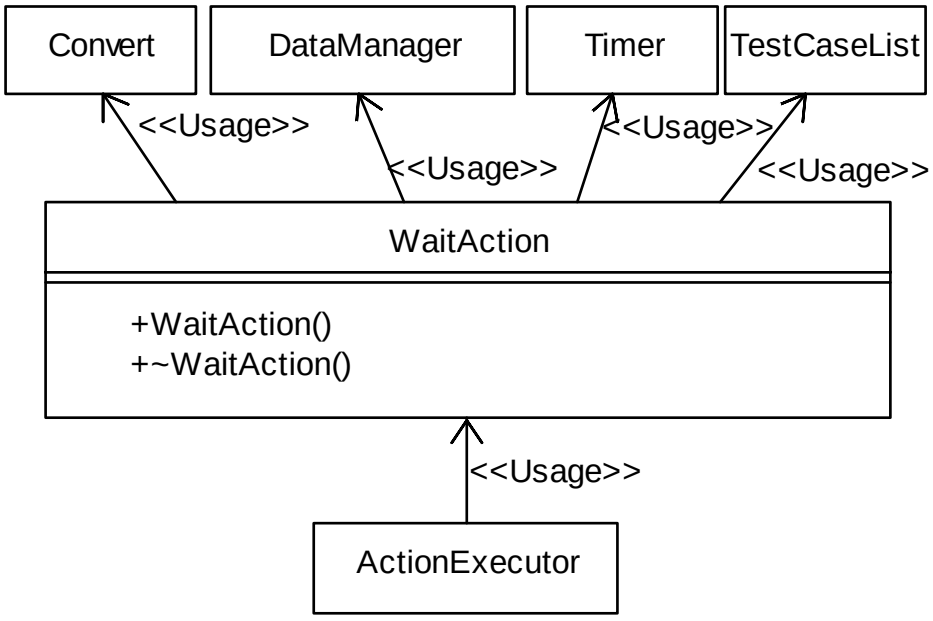


Figure A.16: The wait action class.

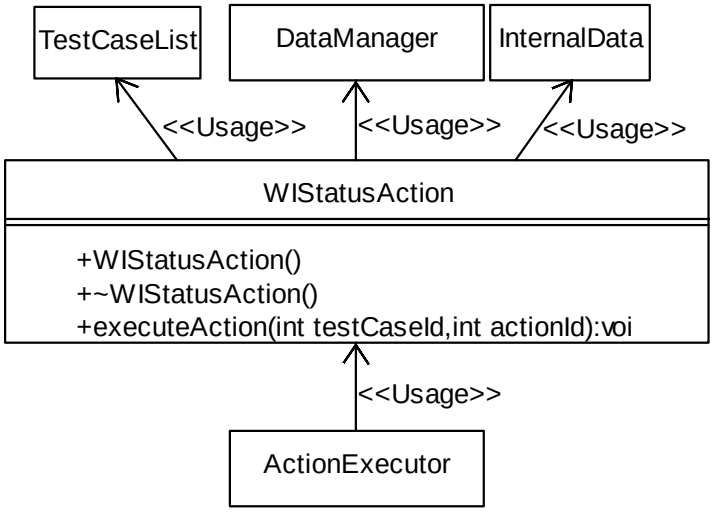


Figure A.17: The WI status action class.

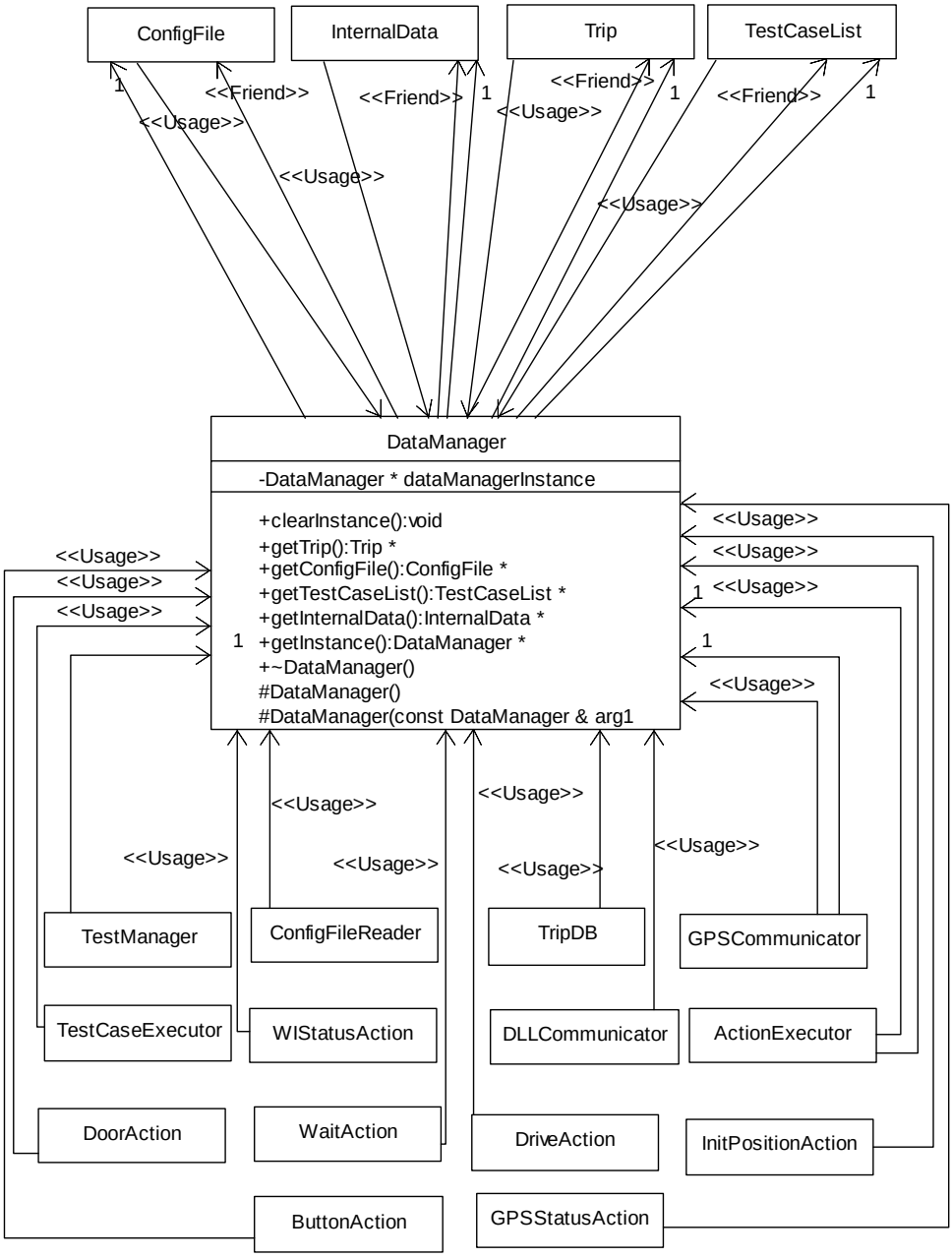


Figure A.18: The data manager class.

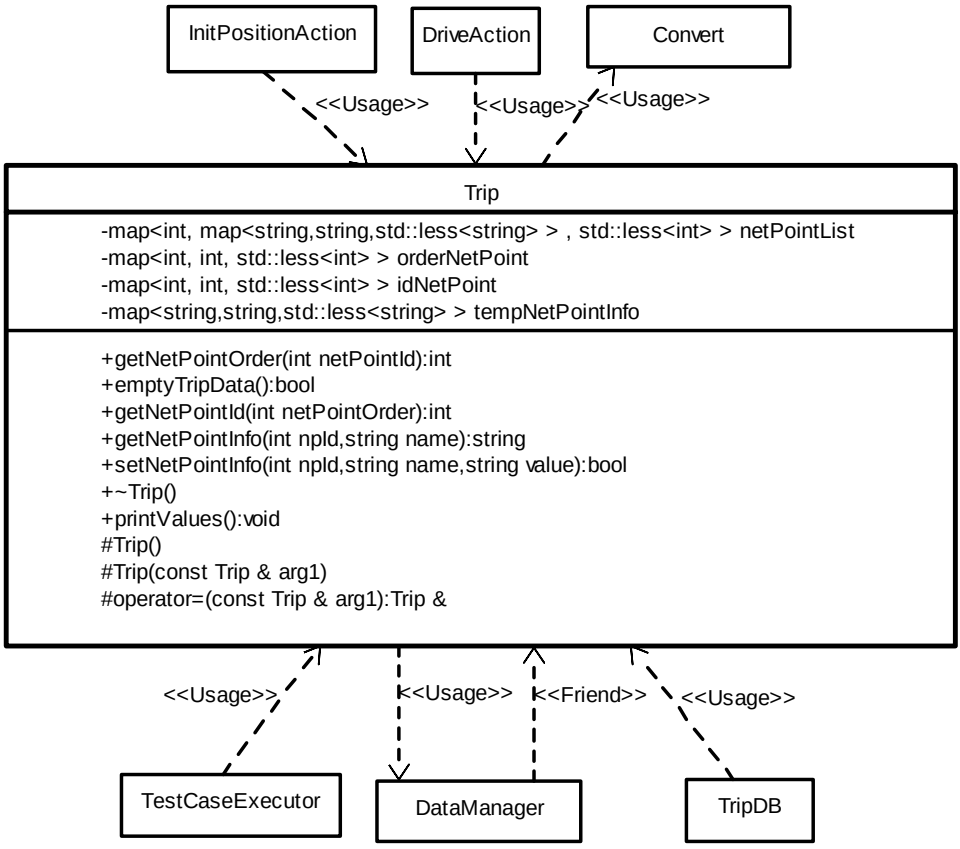


Figure A.19: The trip class.

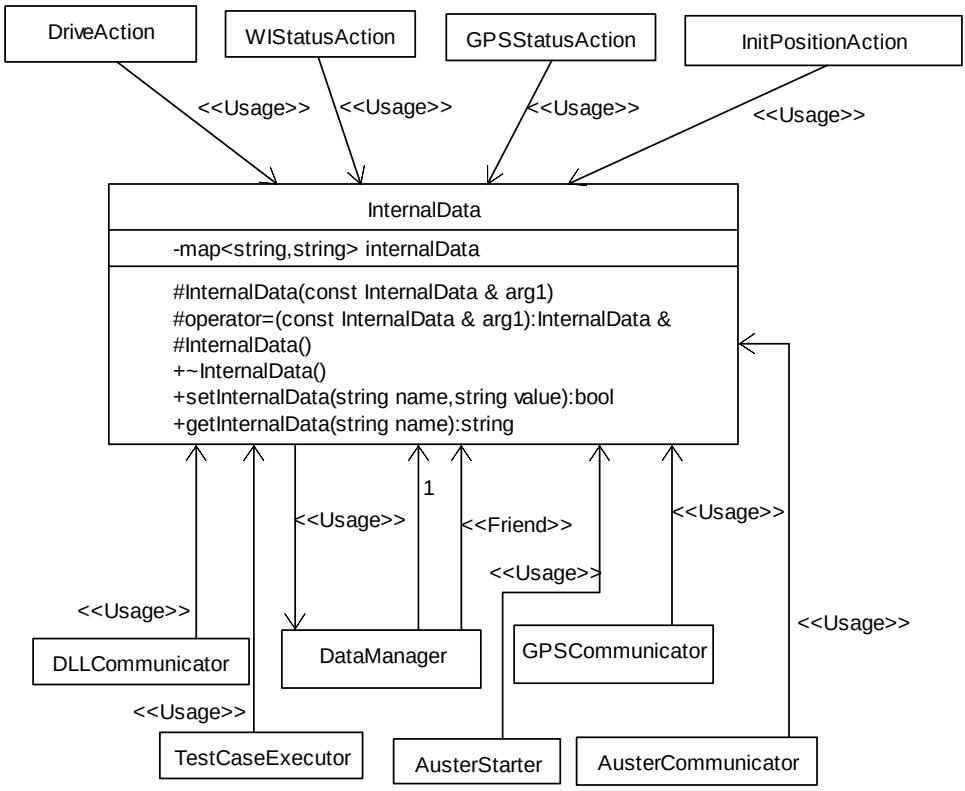


Figure A.20: The internal data class.

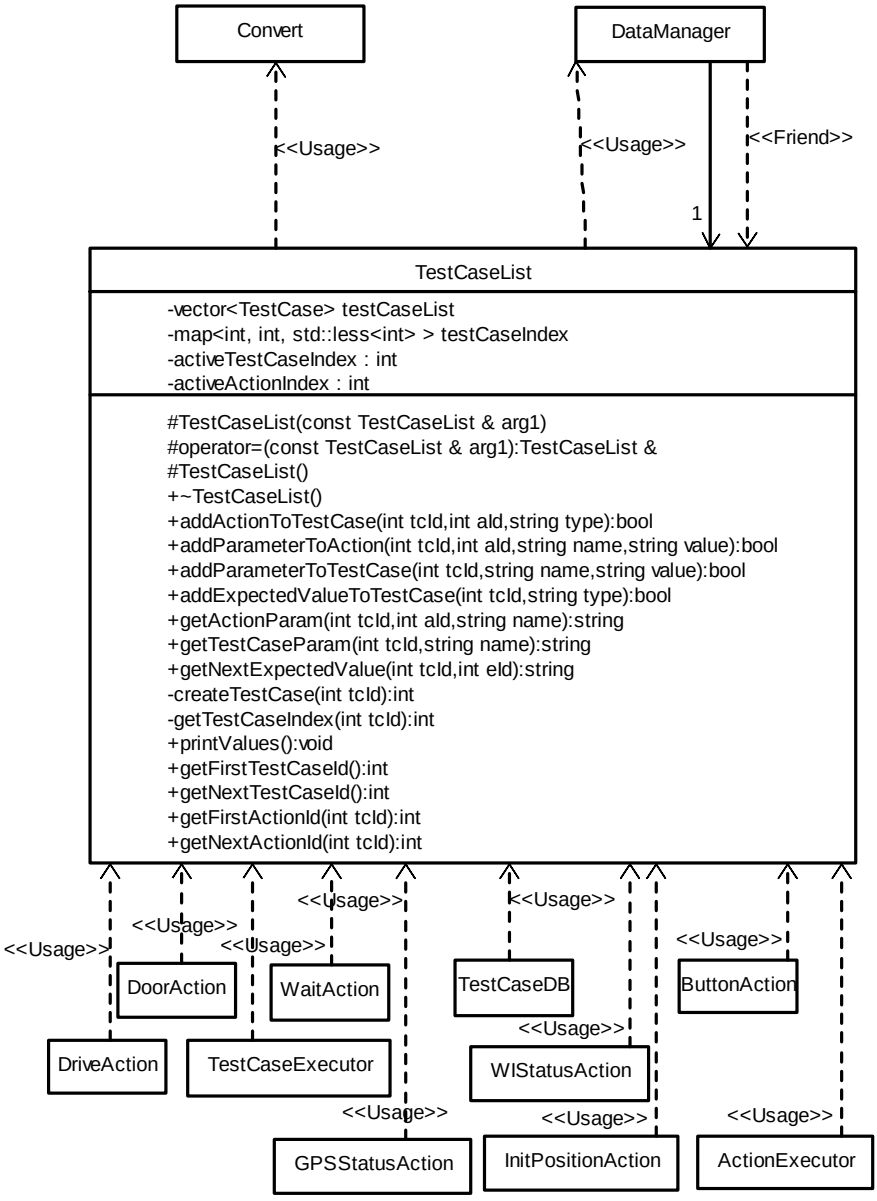


Figure A.21: The test case list class.

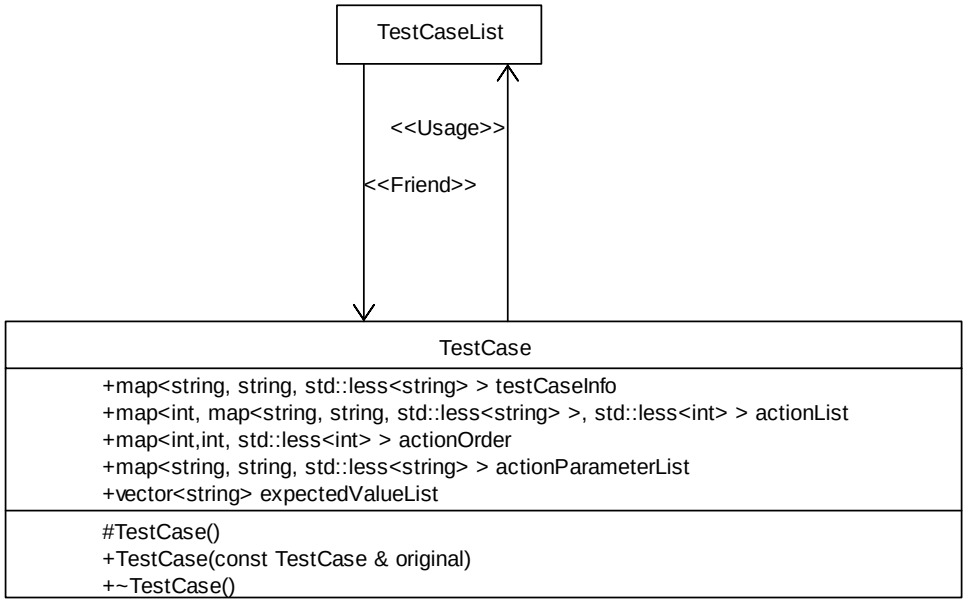


Figure A.22: The test case class.

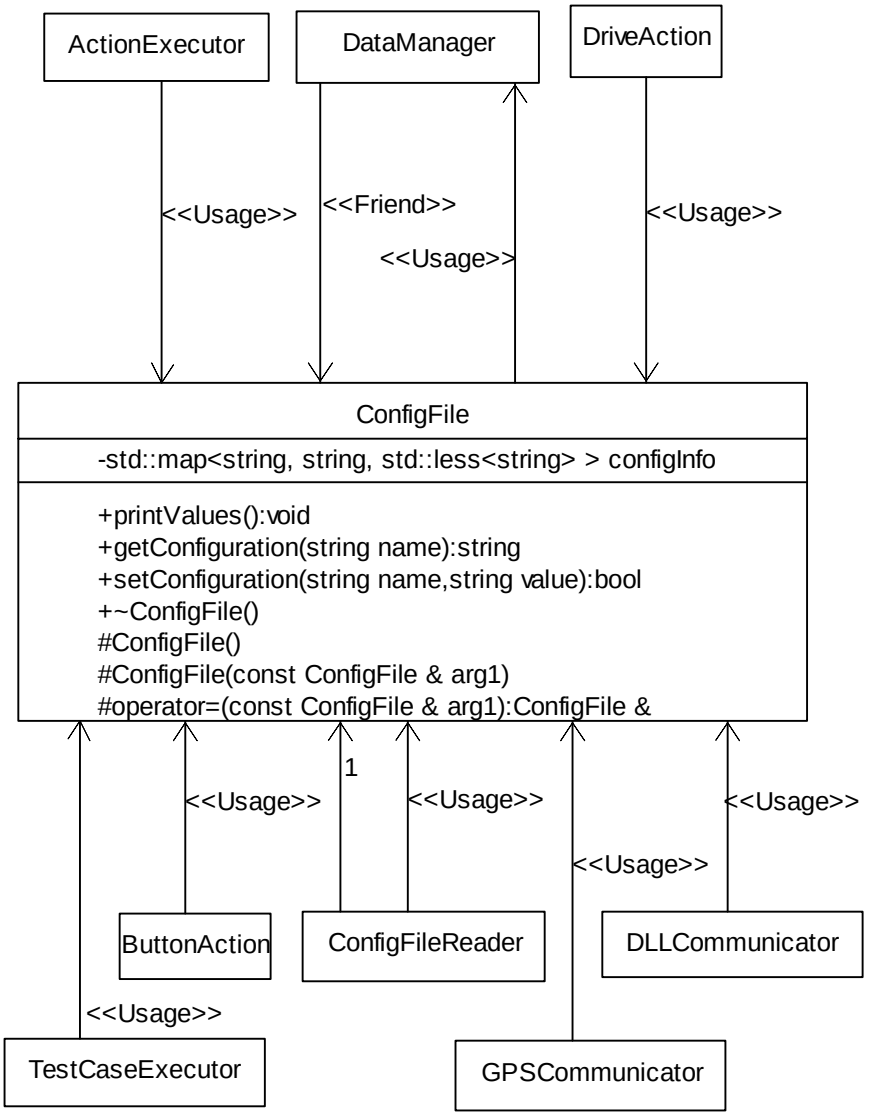


Figure A.23: The config file class.

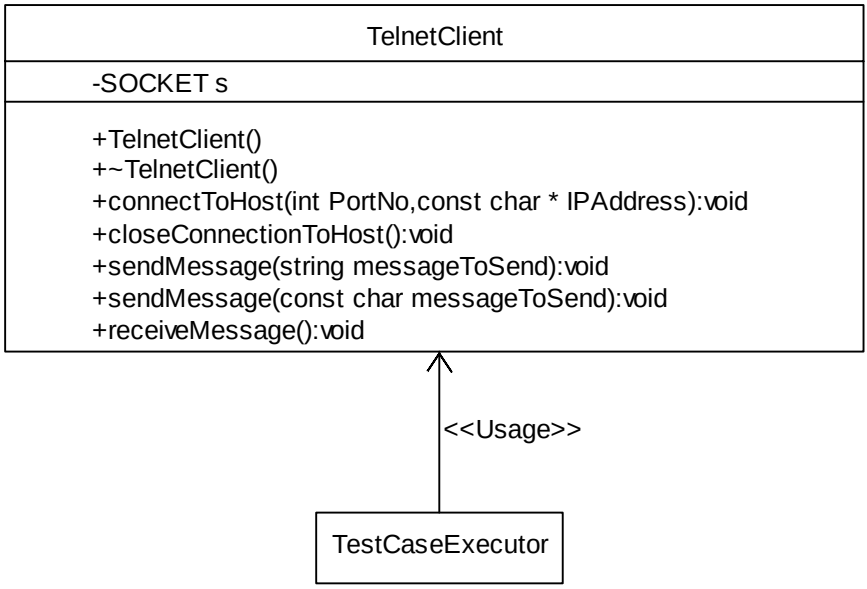


Figure A.24: The Telnet client class.

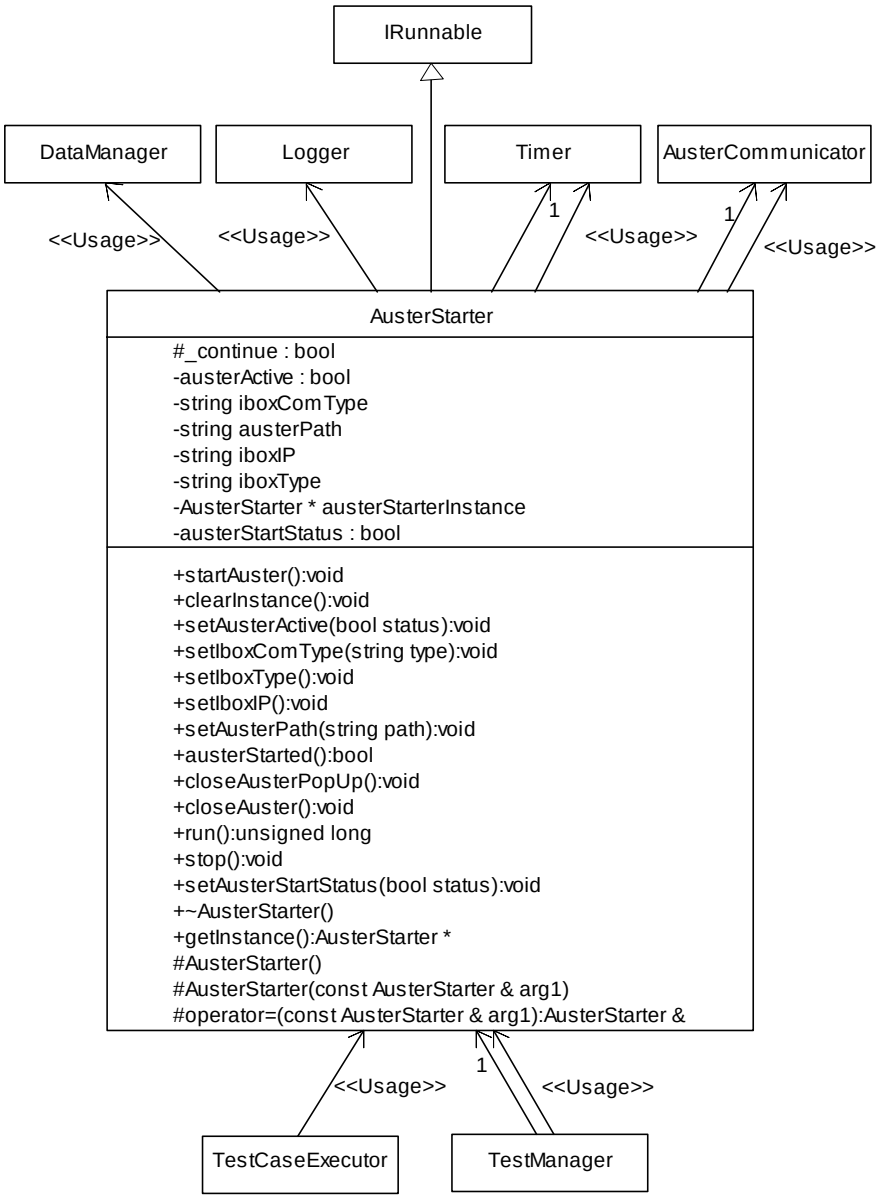


Figure A.25: The auster starter class.

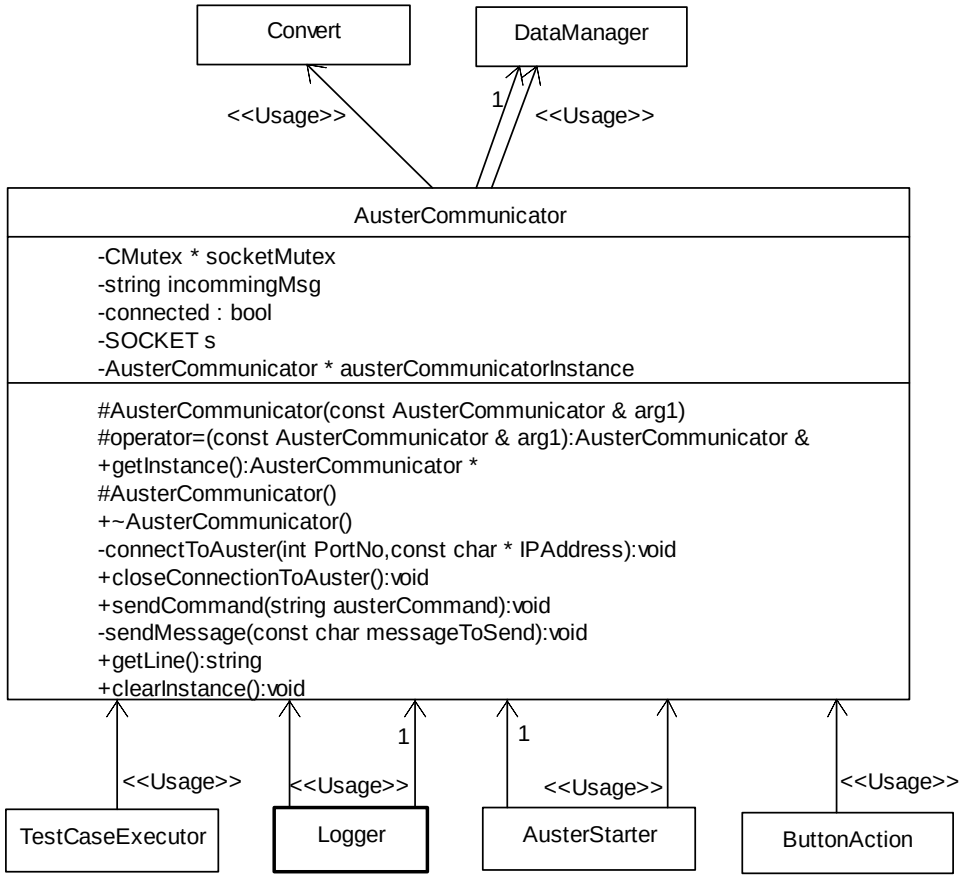


Figure A.26: The auster communicator class.

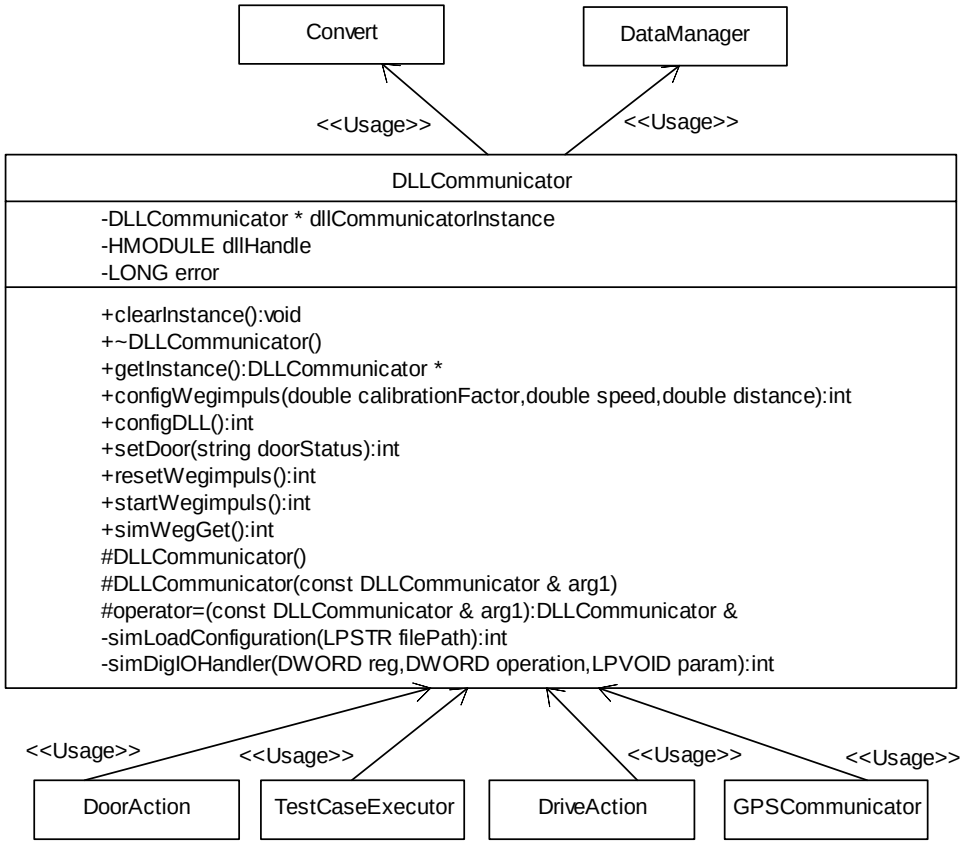


Figure A.27: The DLL communicator class.

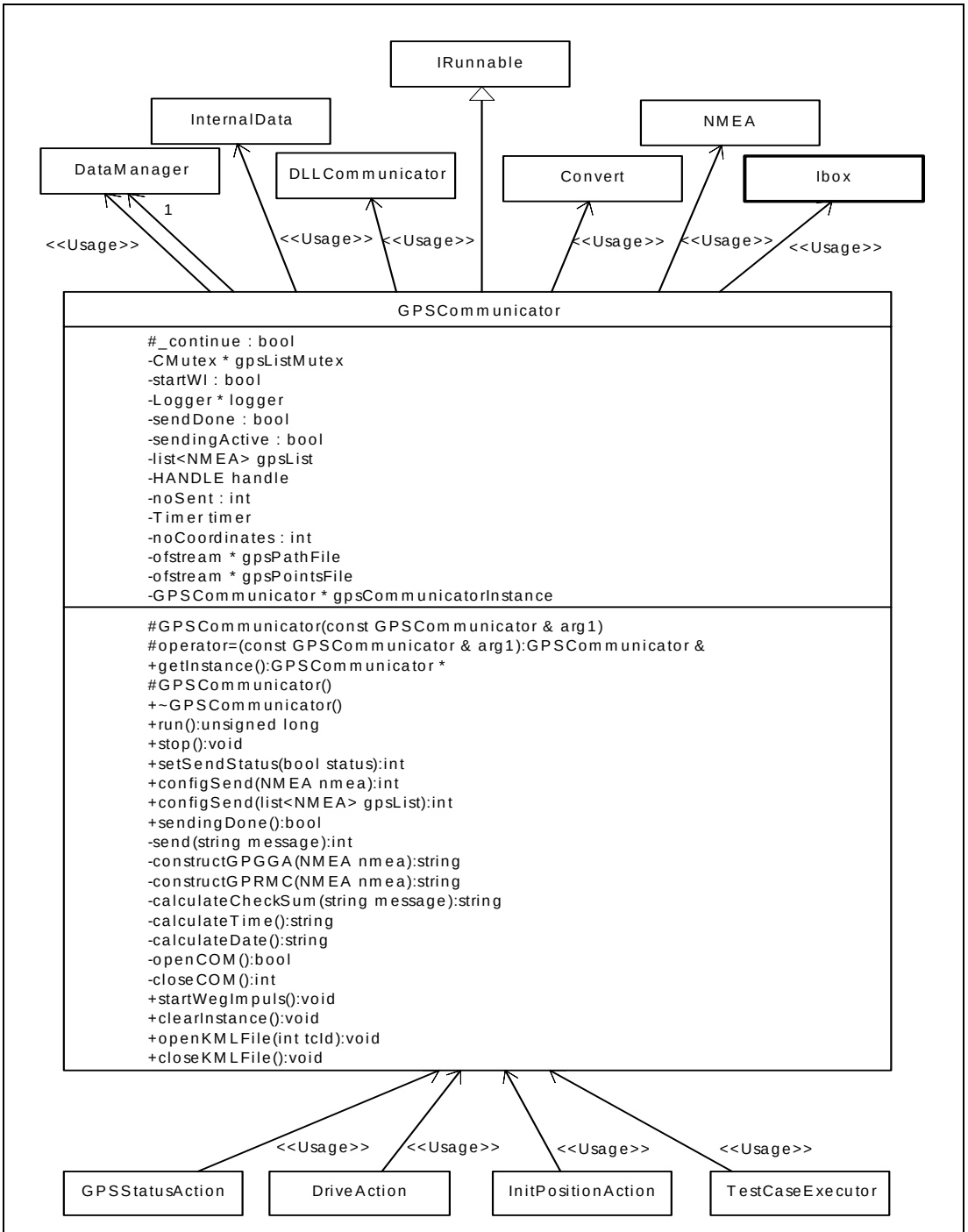


Figure A.28: The GPS communicator class.

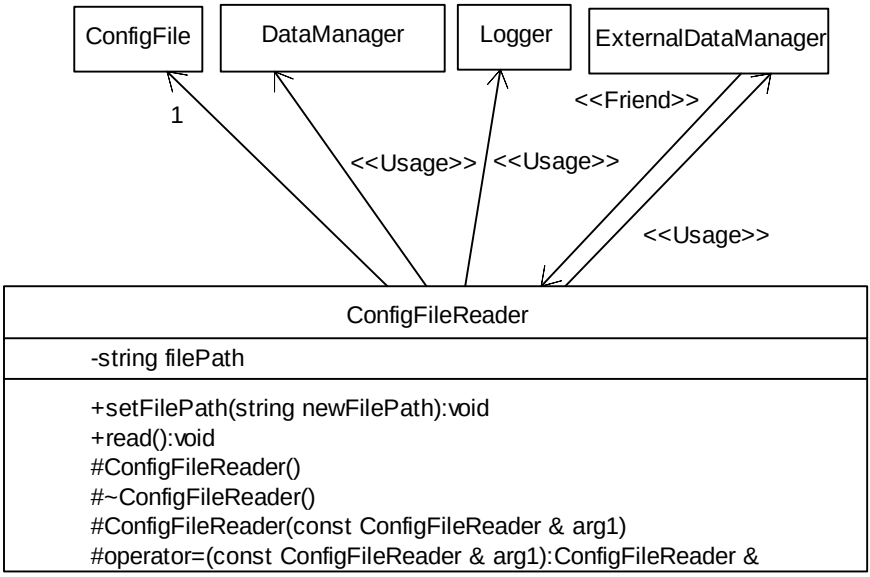


Figure A.29: The config file reader class.

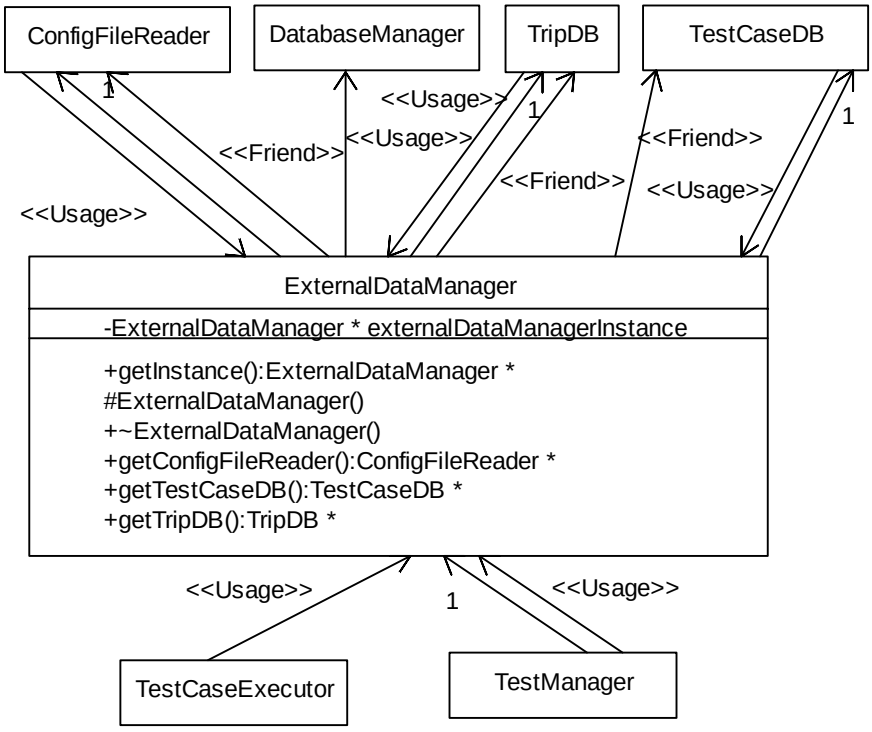


Figure A.30: The external data manager class.

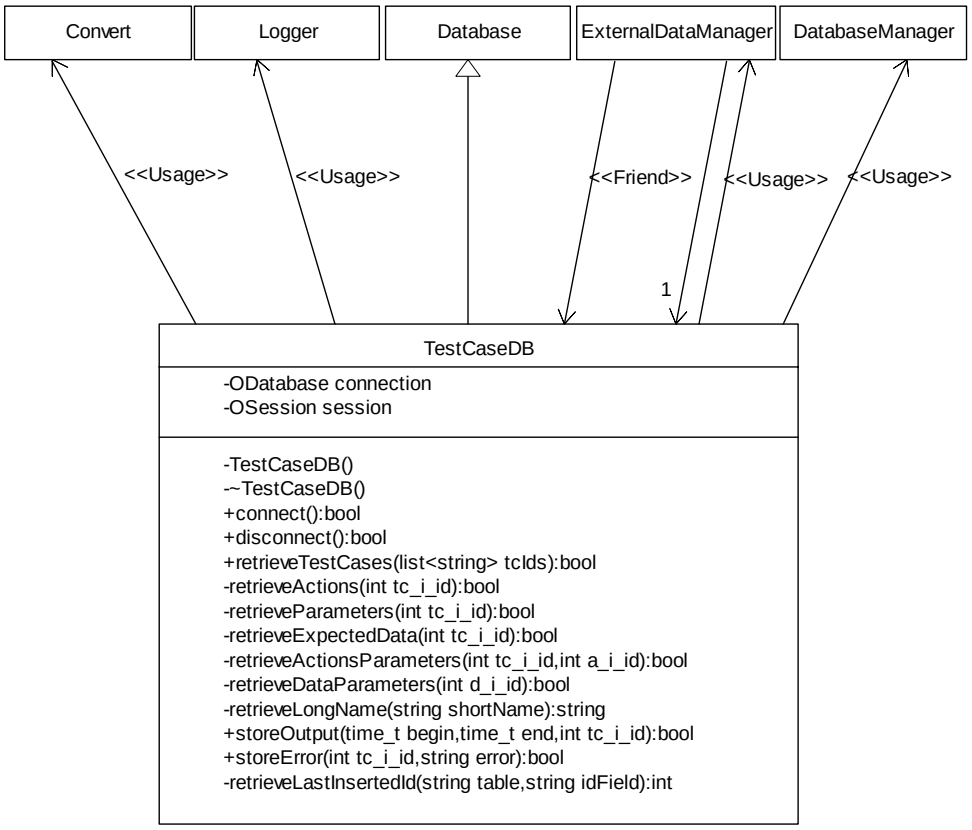


Figure A.31: The test case database class.

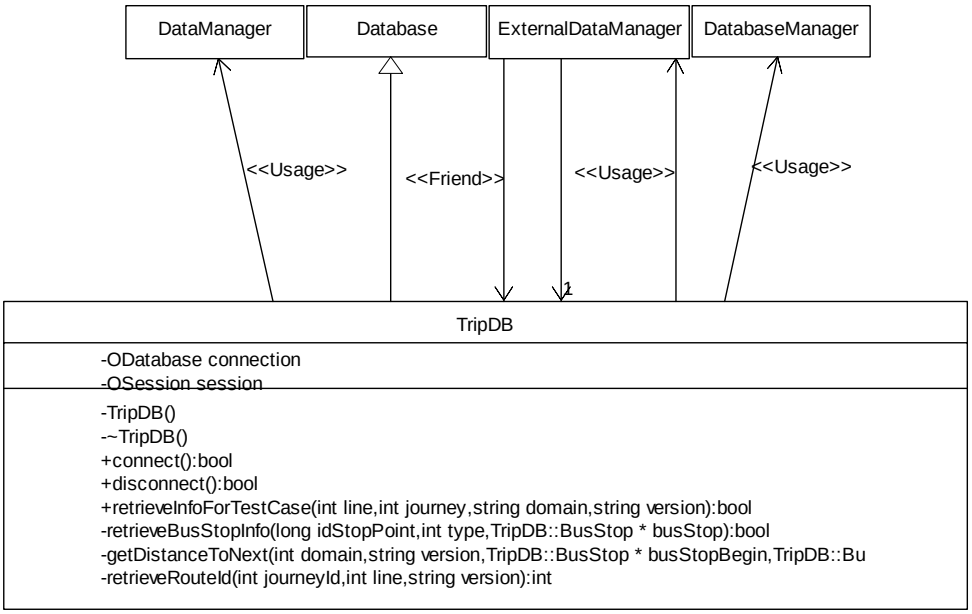


Figure A.32: The trip database class.

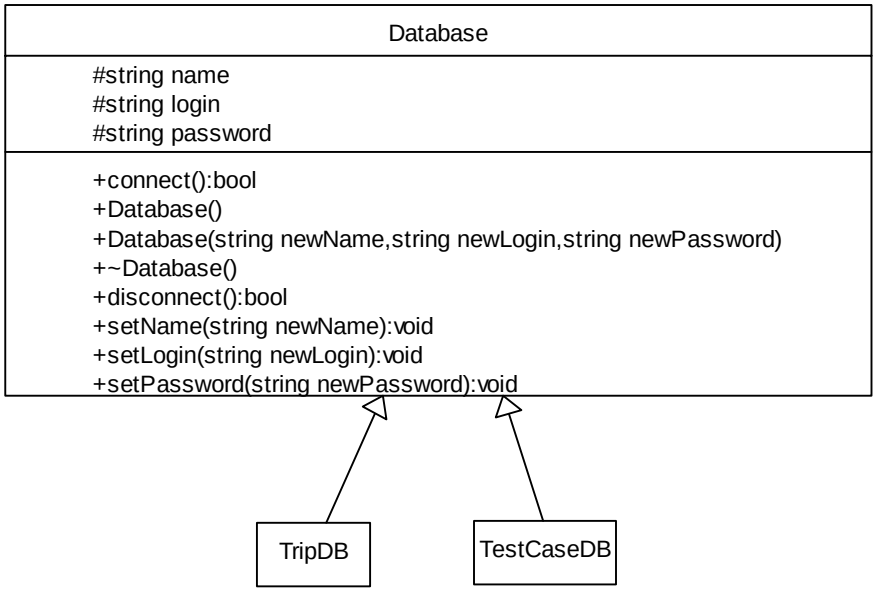


Figure A.33: The database class.

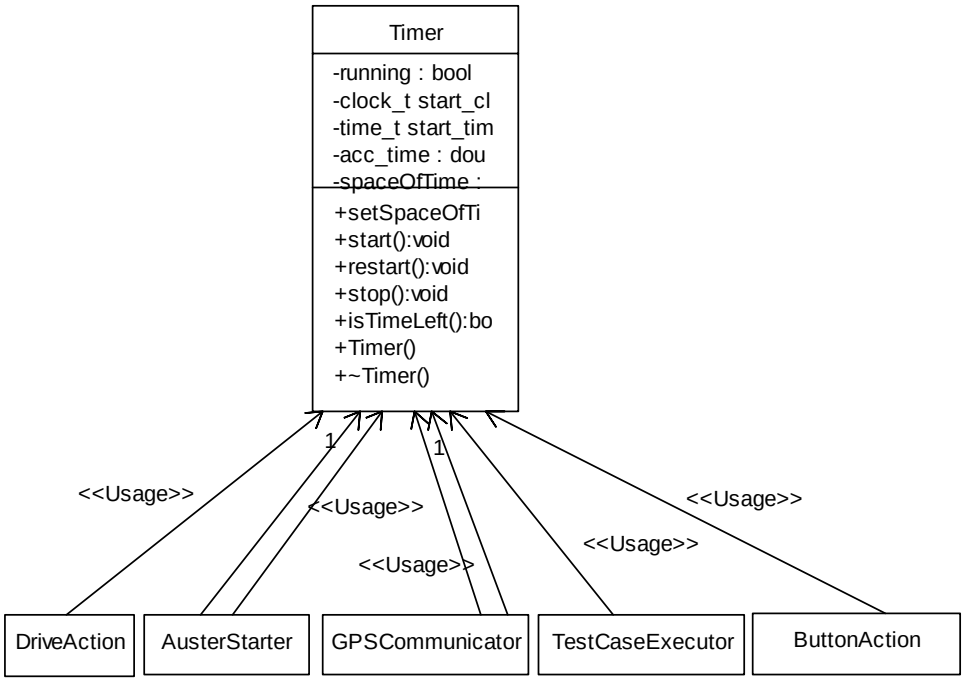


Figure A.34: The timer class.

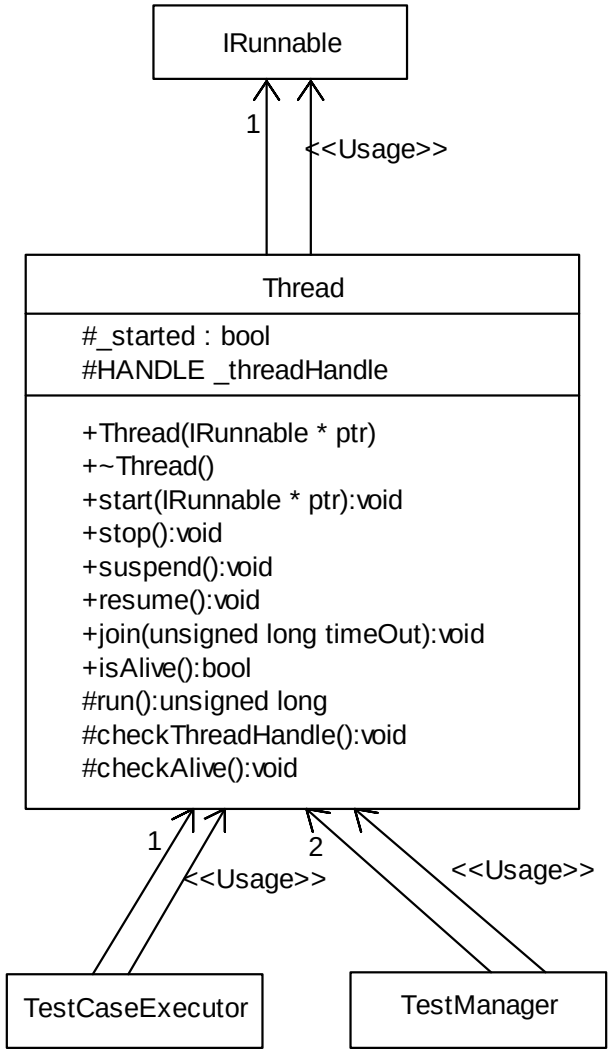


Figure A.35: The thread class.

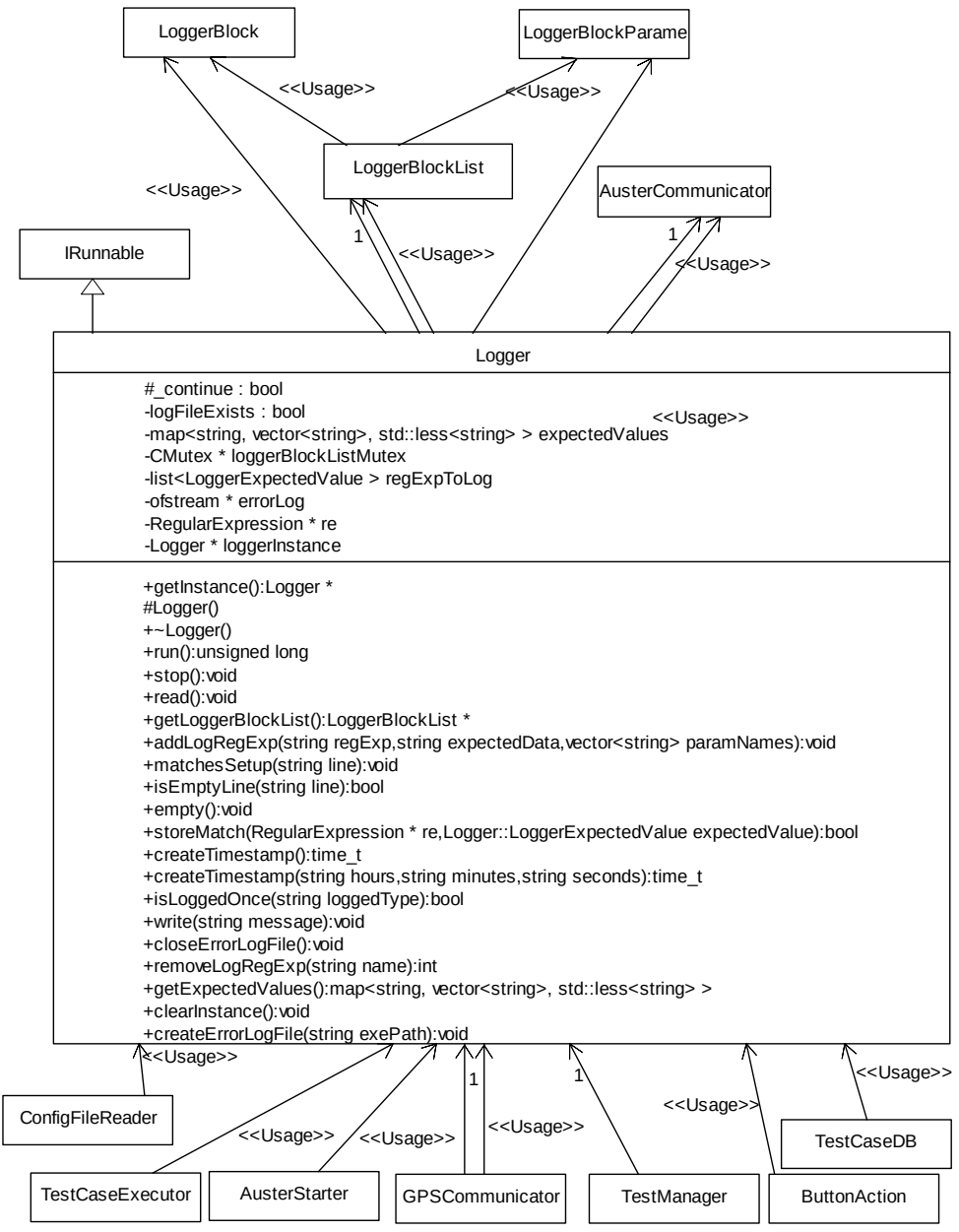


Figure A.36: The logger class.

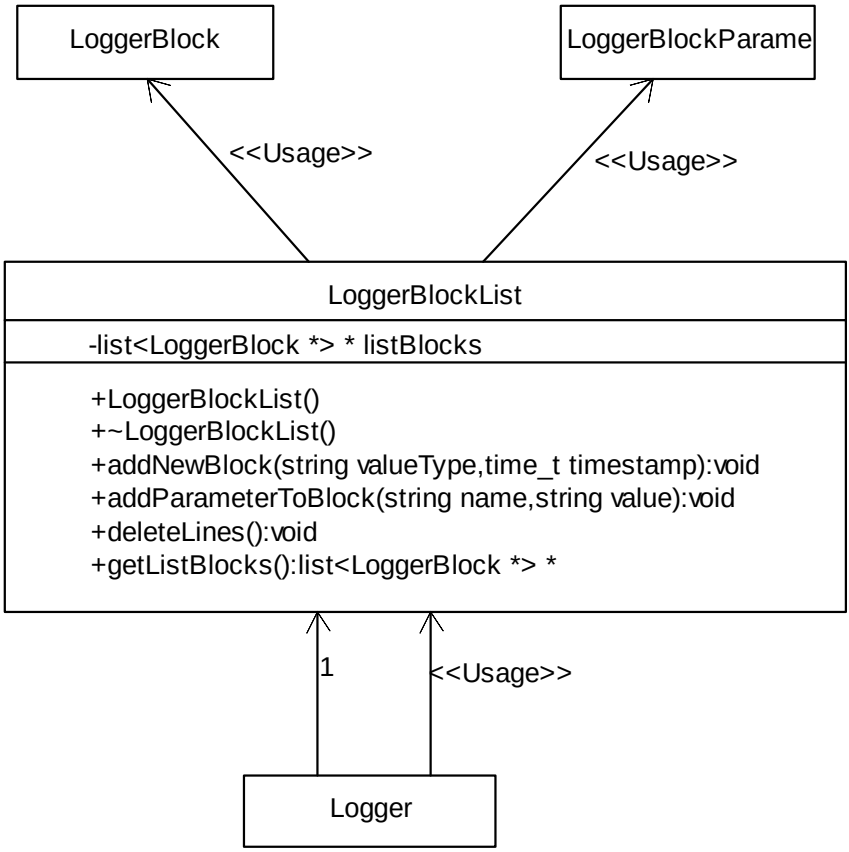


Figure A.37: The logger block list class.

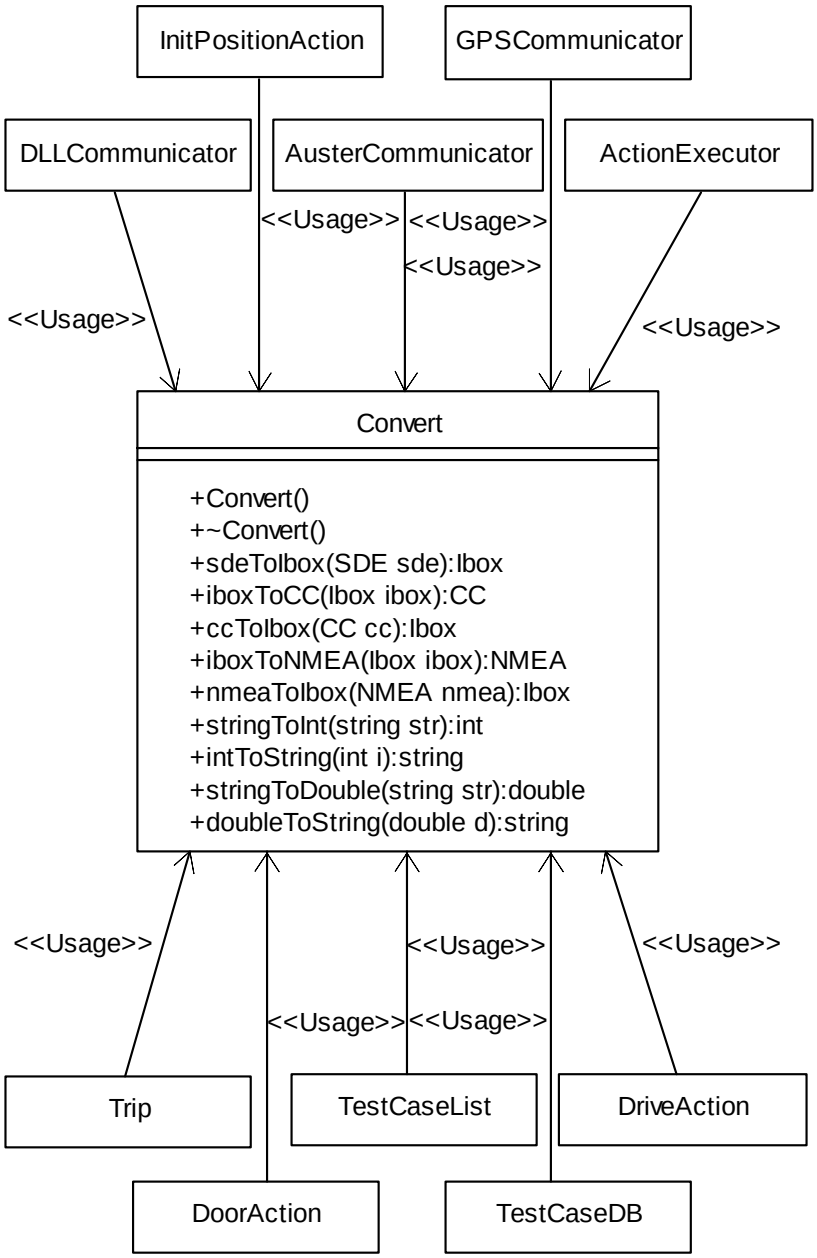


Figure A.38: The convert class.

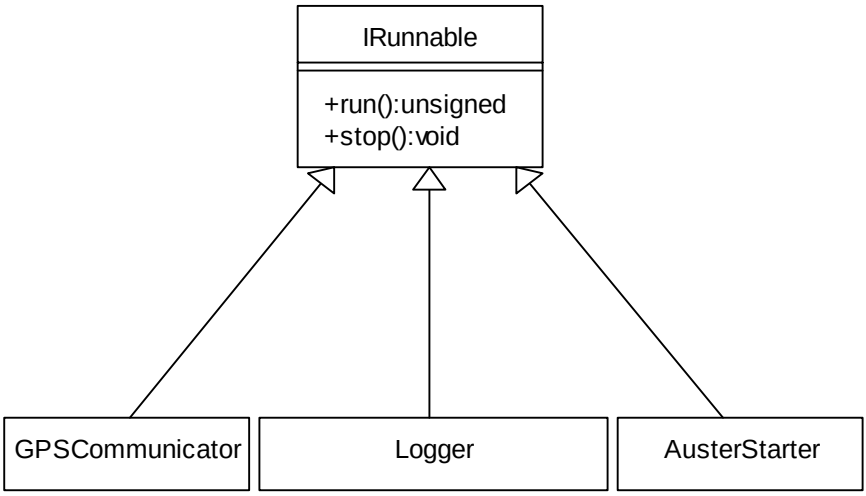


Figure A.39: The IRunnable struct.

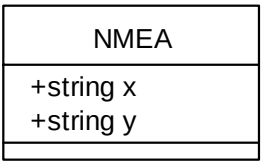


Figure A.40: The NMEA struct.

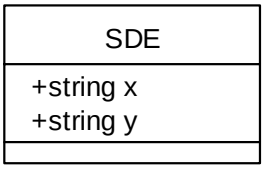


Figure A.41: The SDE struct.

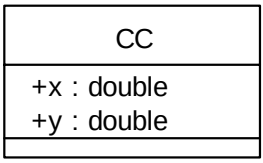


Figure A.42: The CC struct.

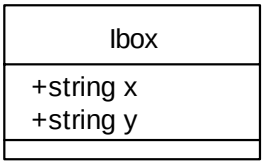


Figure A.43: The i.box struct

A.1.3 Activity diagrams

The following figures show the activities and the flow of the program, a little bit simplified.

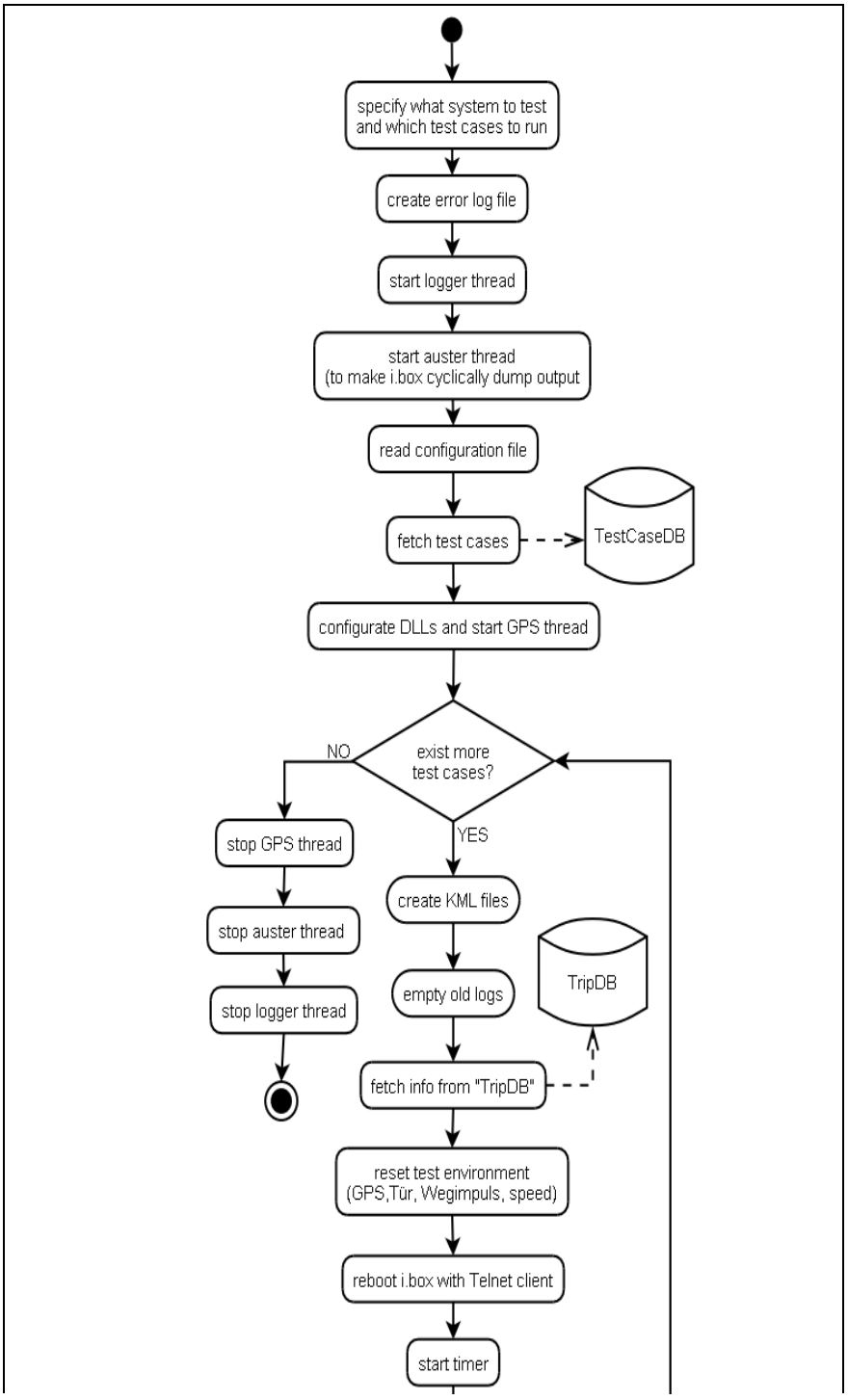
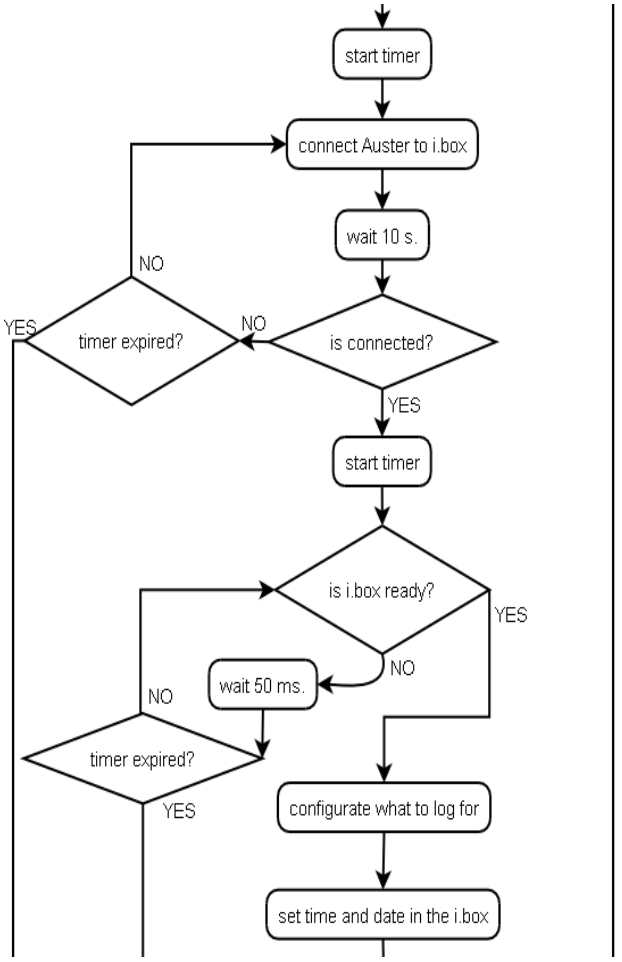

specify what system to test and which test cases to run
create error log file
start logger thread
start auster thread (to make i.box cyclically dump output
read configuration file
fetch test cases
TestCaseDB
configurate DLLs and start GPS thread
exist more test cases?
NO
YES
stop GPS thread
stop auster thread
stop logger thread
create KML files
empty old logs
TripDB
fetch info from "TripDB"
reset test environment (GPS,Tür, Wegimpuls, speed)
reboot i.box with Telnet client
start timer

Figure A.44: Activity diagram for the main flow in test performance.



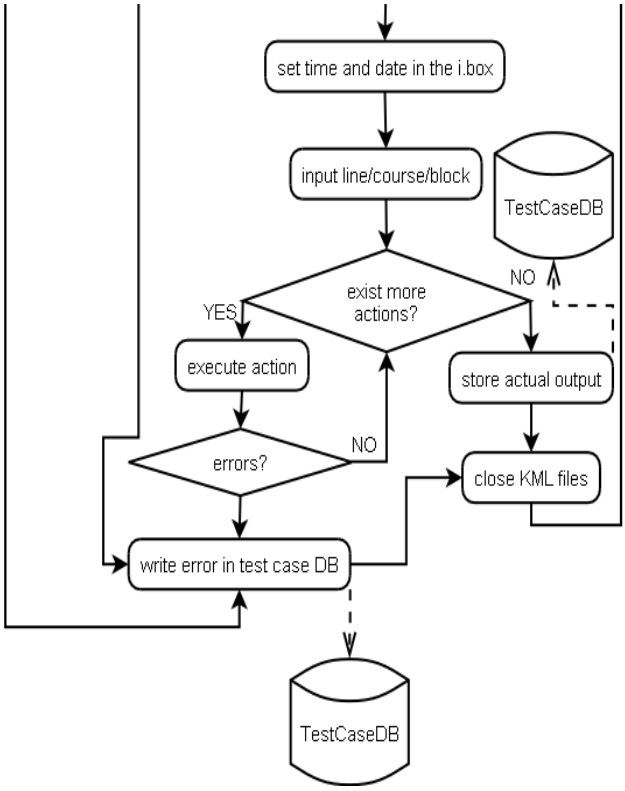
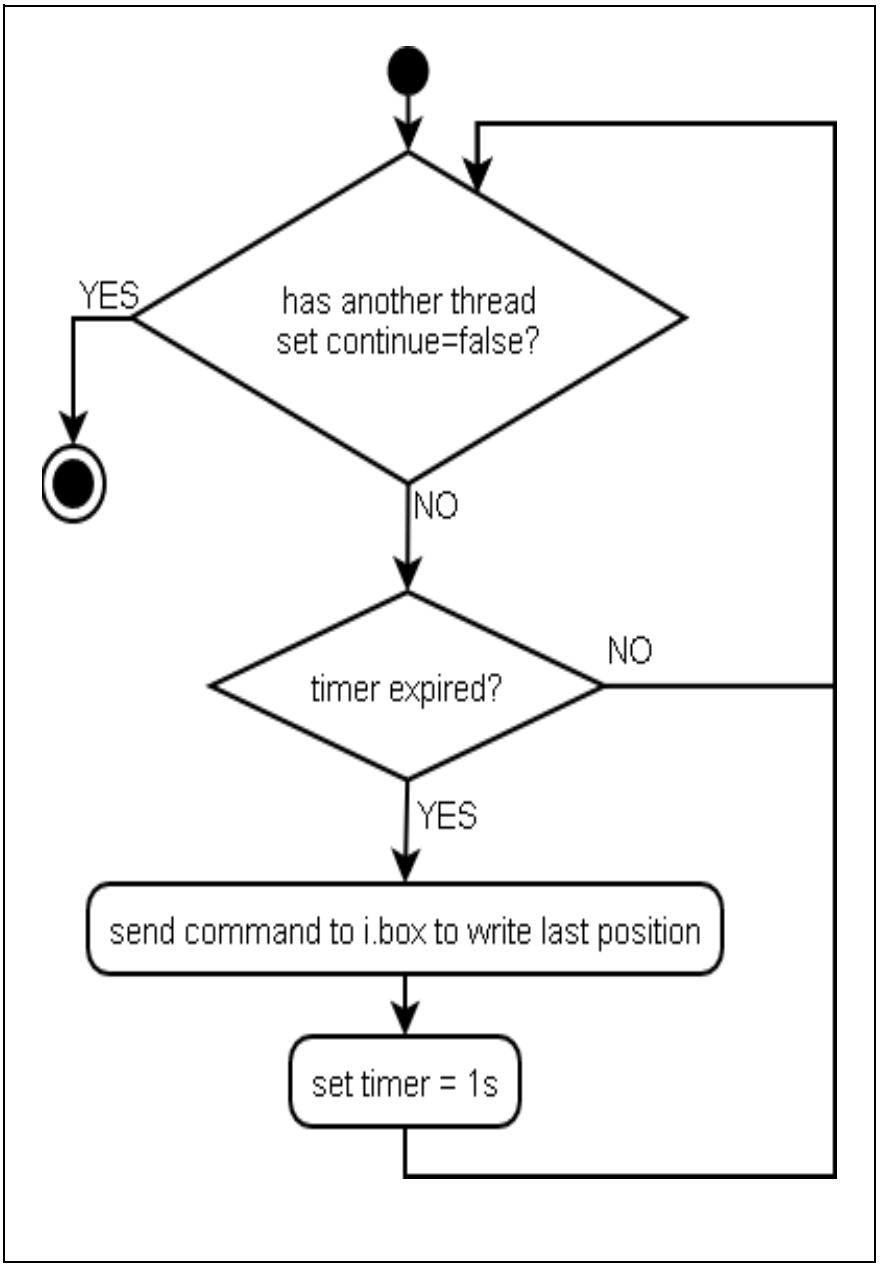
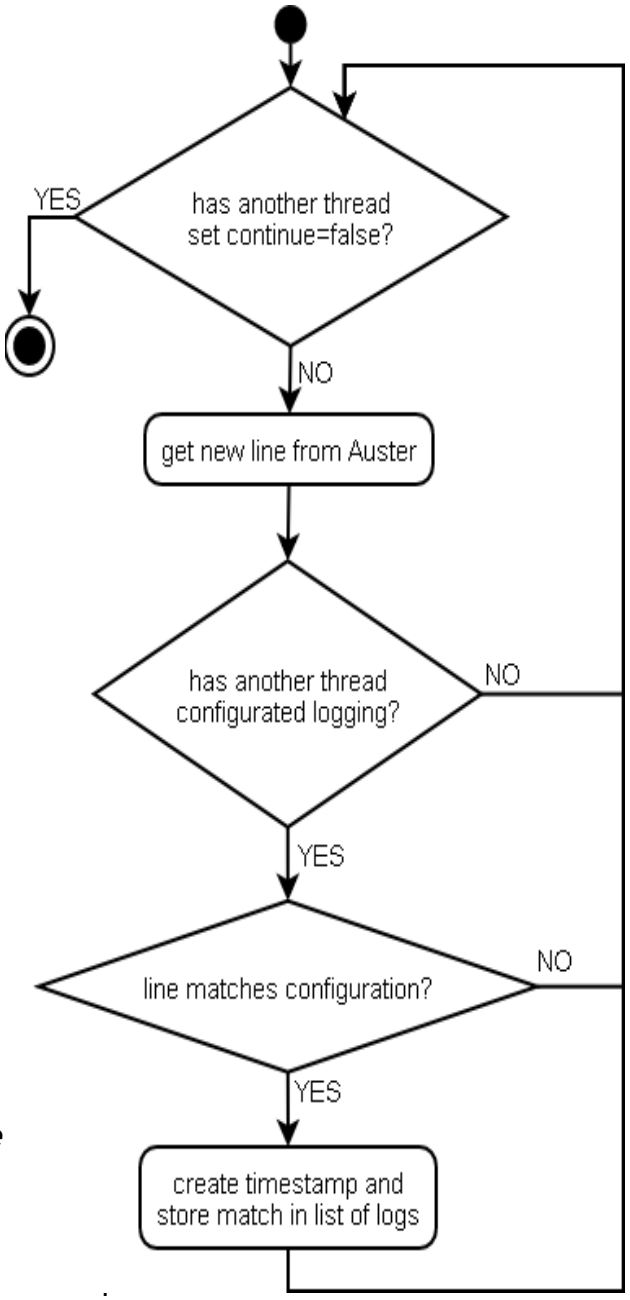


Figure A.45: Activity diagram for the Auster thread.


has another thread
set continue=false?
YES
NO
timer expired?
NO
YES
send command to i.box to write last position
set timer = 1s



A.4 Figure 46

vity diagram for the logger thread.

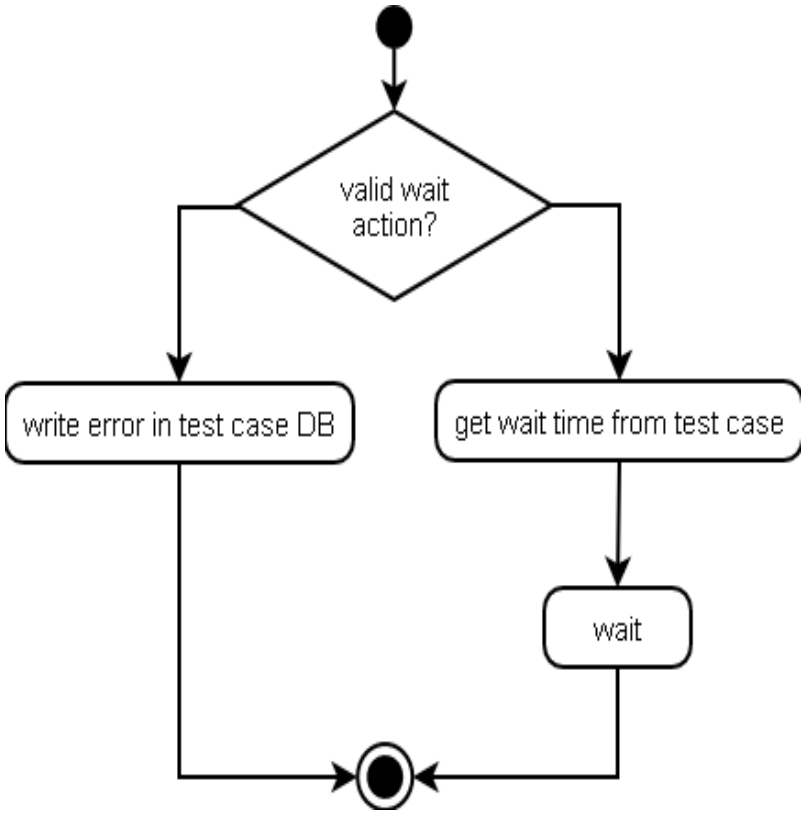


Figure A.47: Activity diagram for the action wait.

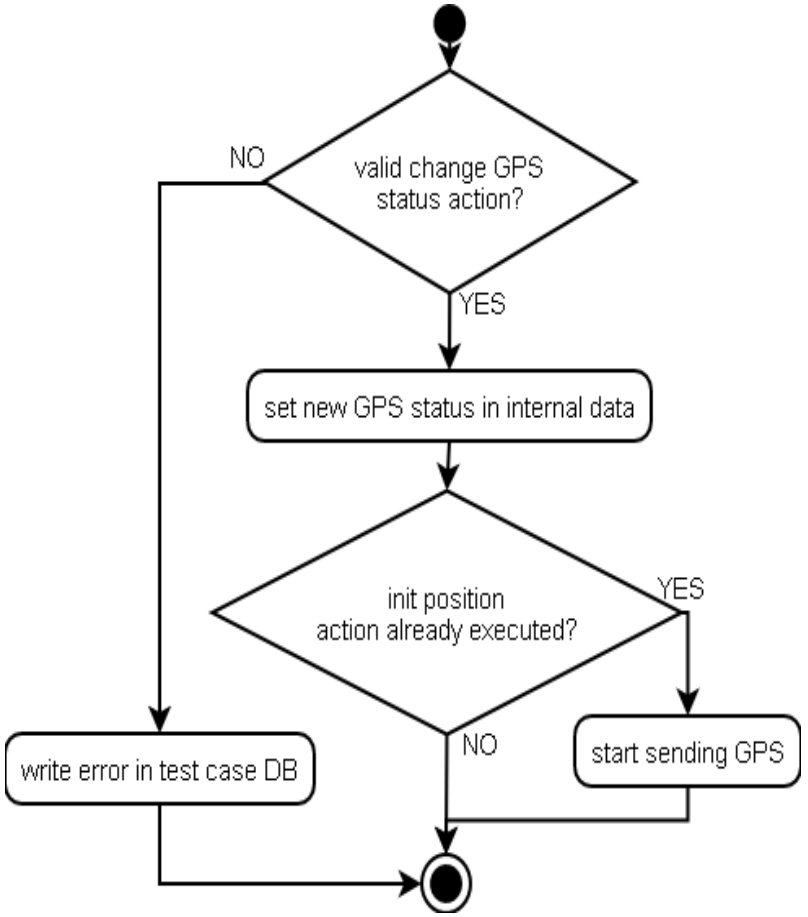


Figure A.48: Activity diagram for the action turn on or shut off the GPS.

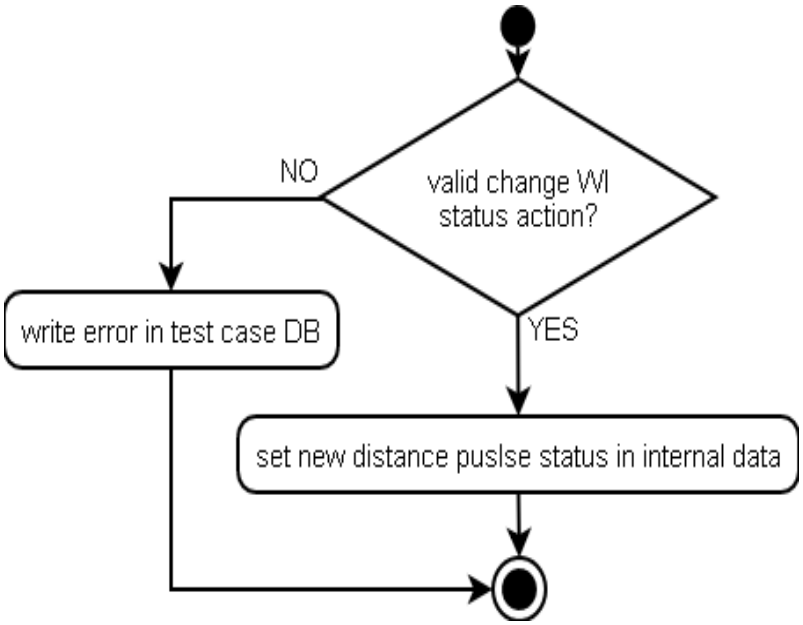
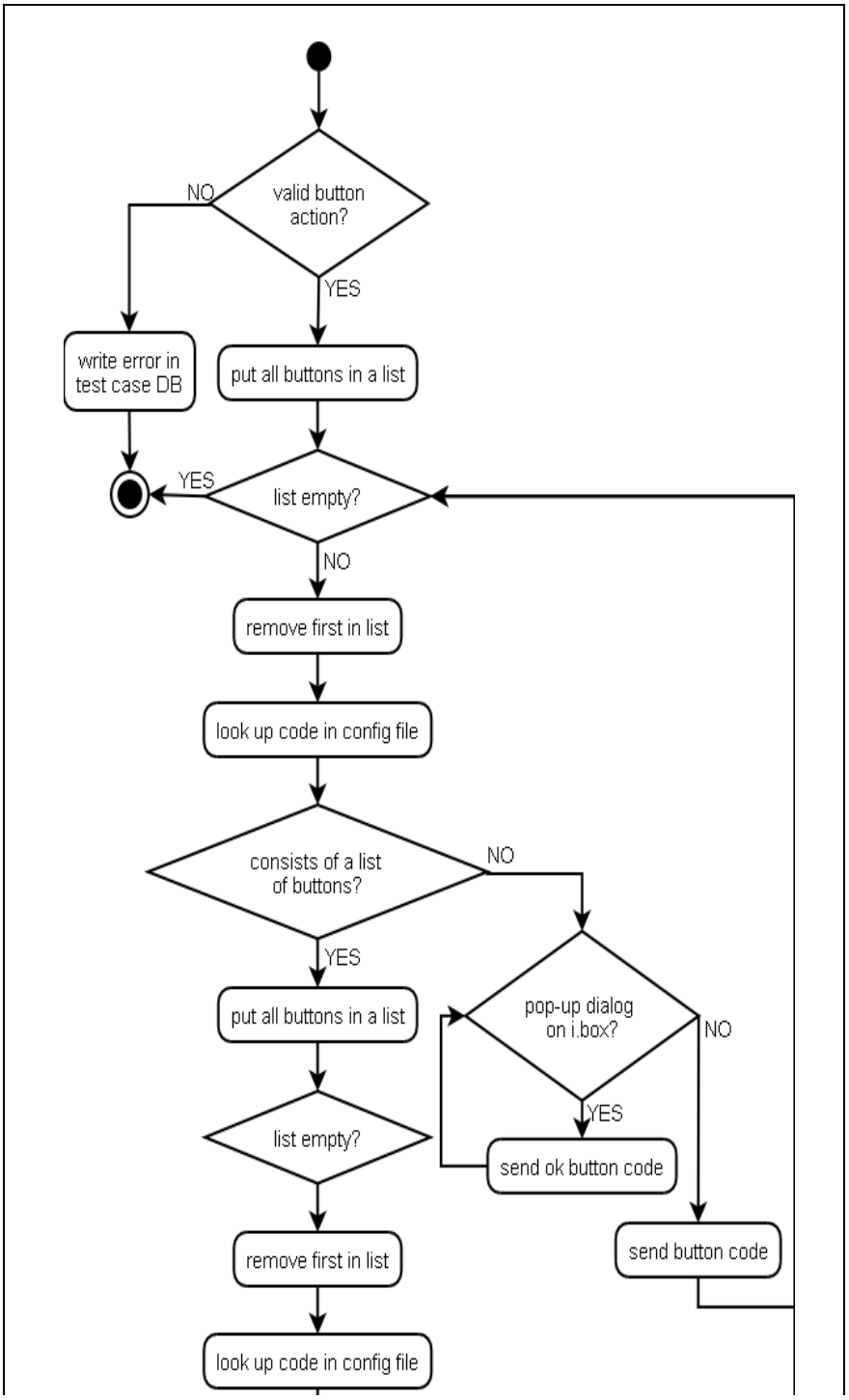


Figure A.49: Activity diagram for the action turn on or shut off the distance pulse.


valid button action?
NO
YES
write error in test case DB
put all buttons in a list
list empty?
YES
NO
remove first in list
look up code in config file
consists of a list of buttons?
NO
YES
put all buttons in a list
pop-up dialog on i.box?
NO
YES
list empty?
send ok button code
send button code
remove first in list
look up code in config file

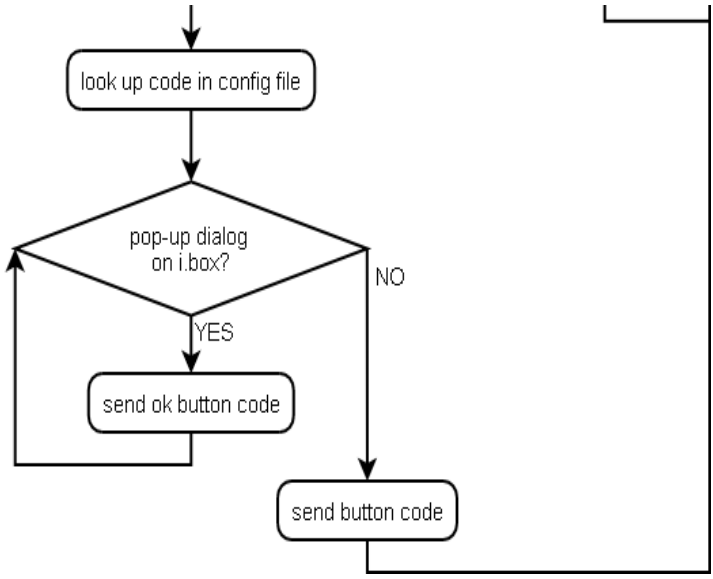


Figure A.50: Activity diagram for the action push buttons.

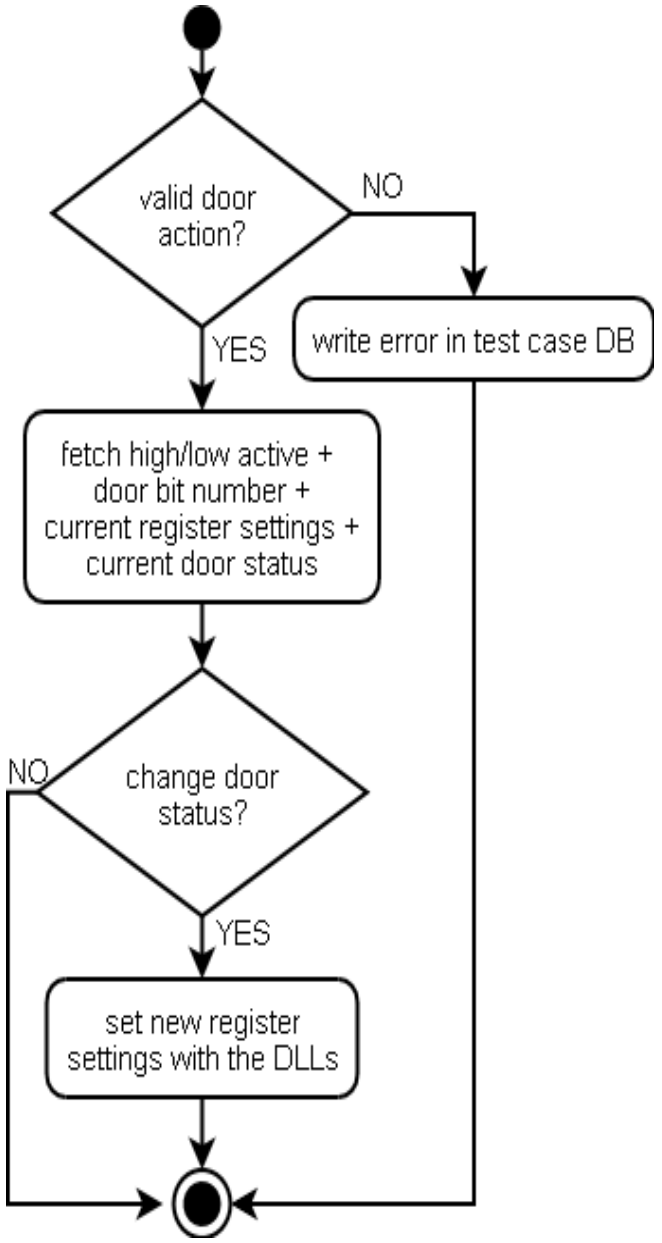
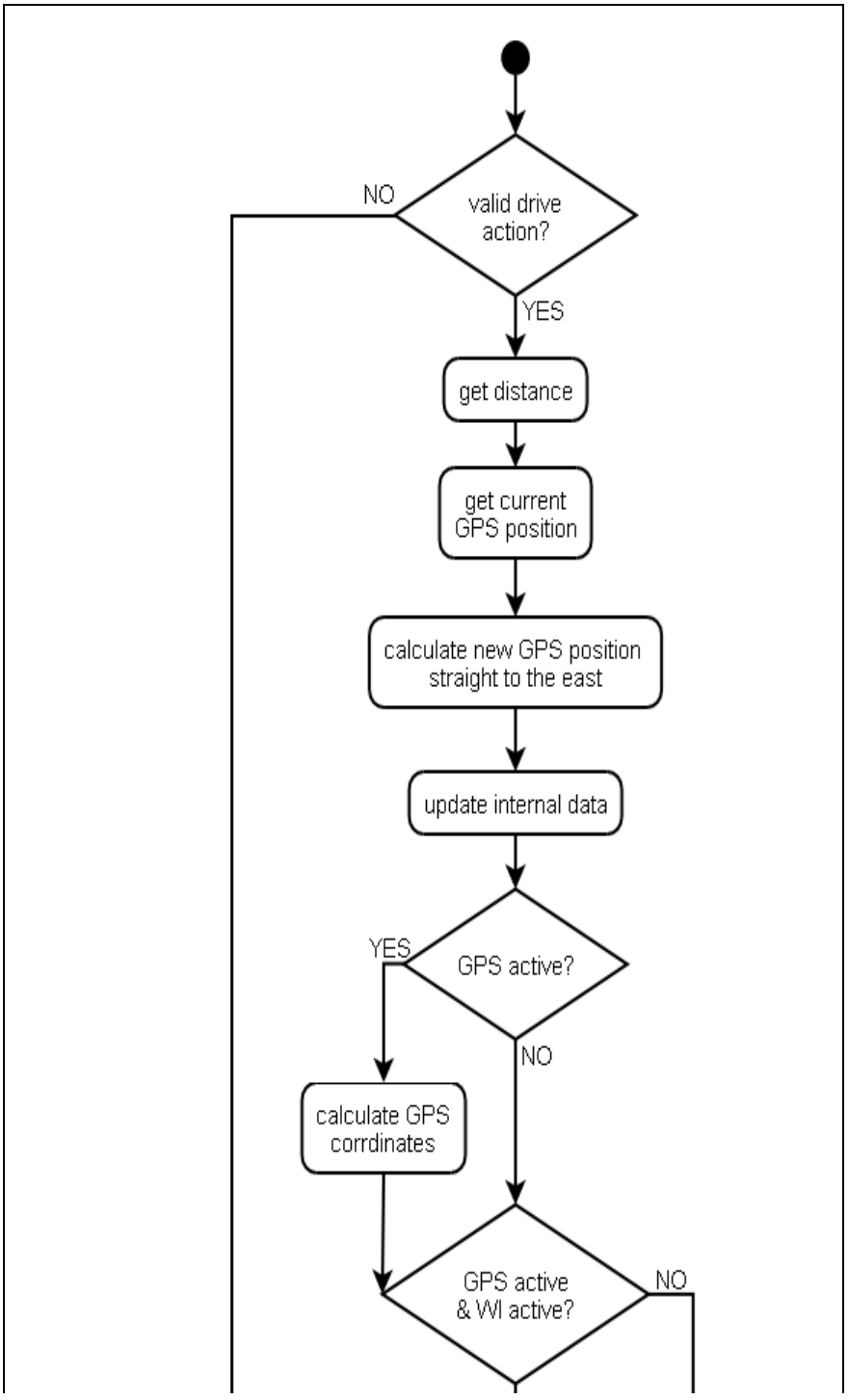
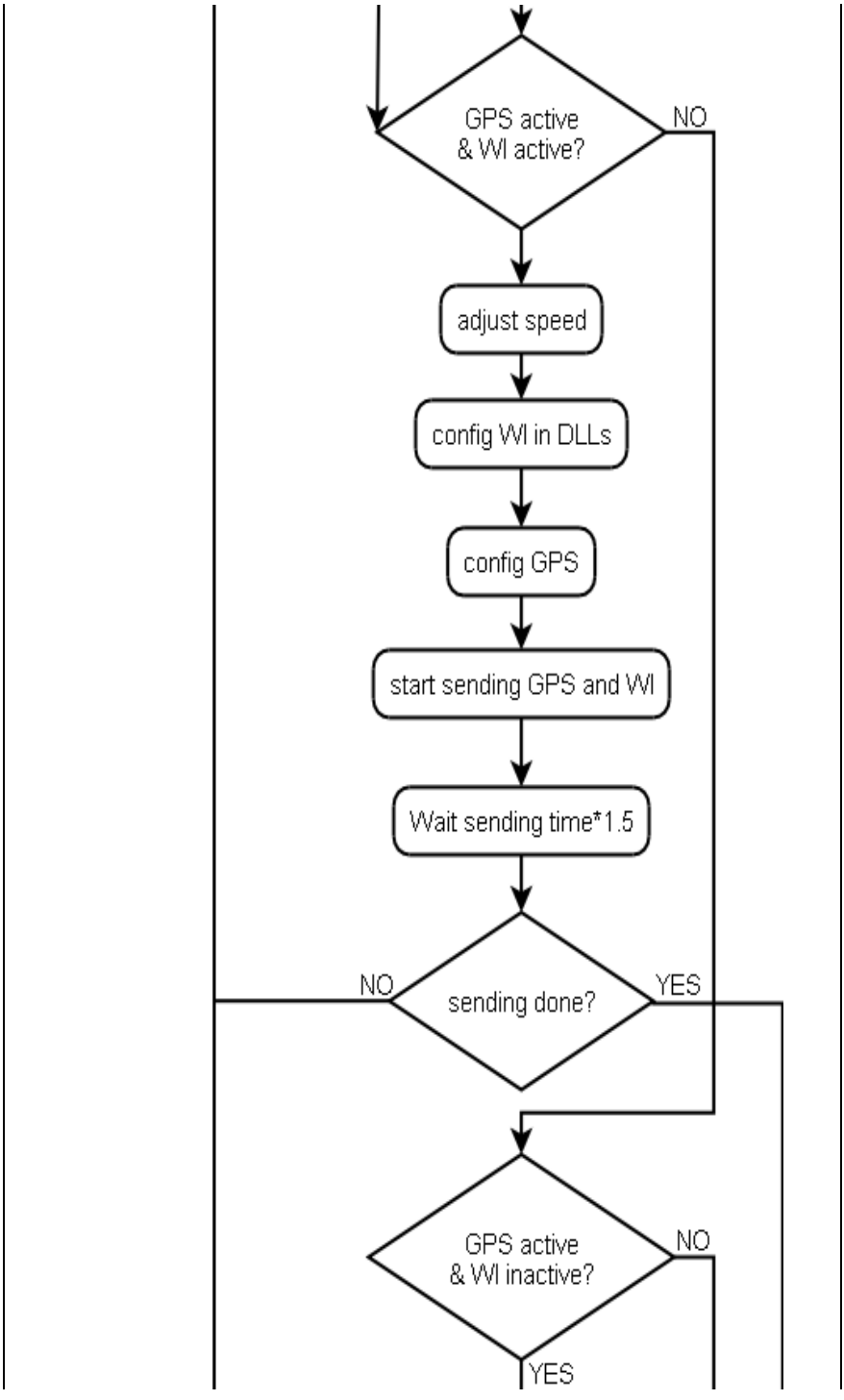
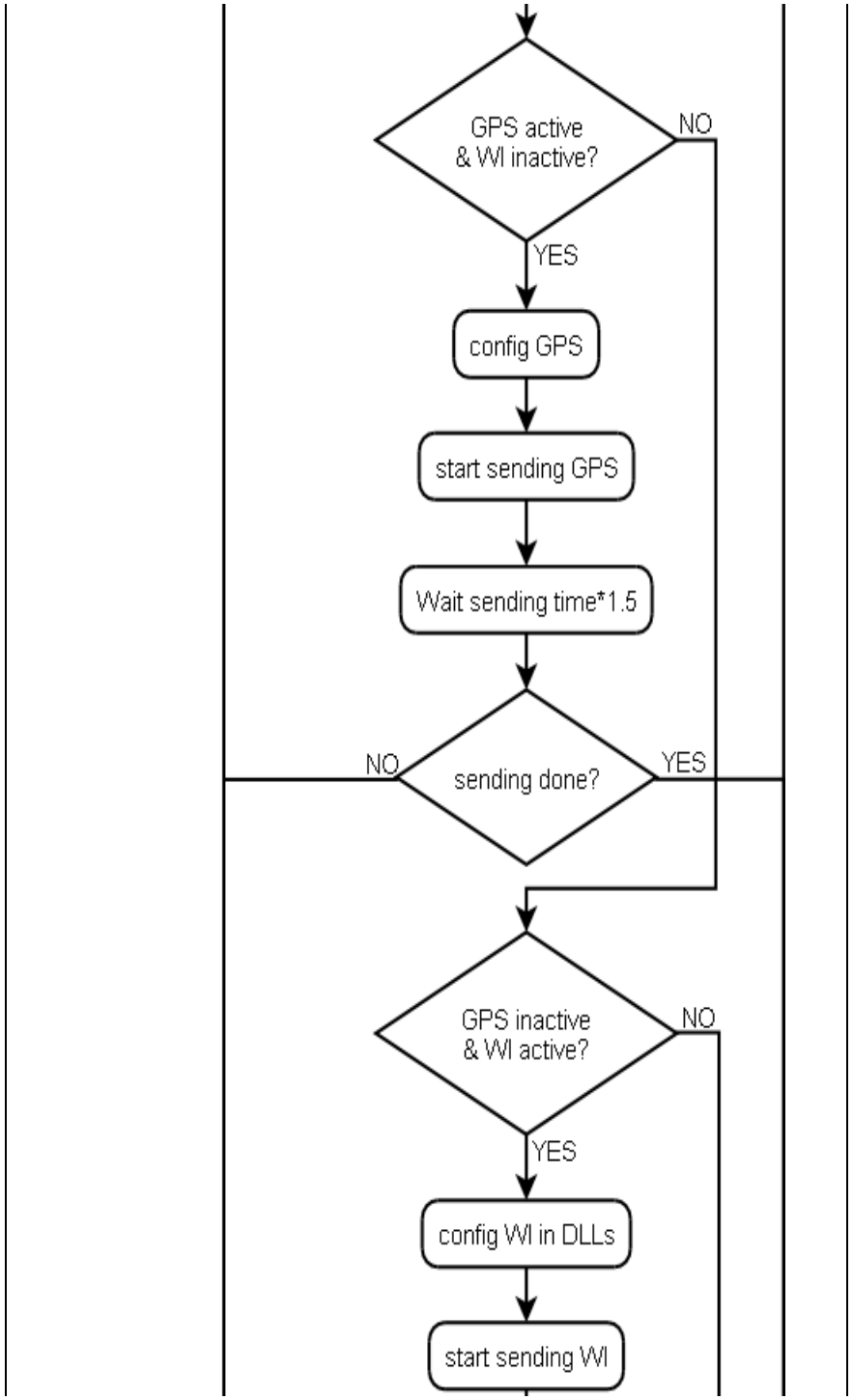


Figure A.51: Activity diagram for the action open/close door.


valid drive action?
NO
YES
get distance
get current GPS position
calculate new GPS position straight to the east
update internal data
GPS active?
YES
NO
calculate GPS corrdinates
GPS active & WI active?
NO


GPS active
& WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI
Wait sending time*1.5
NO
sending done?
YES
GPS active
& WI inactive?
NO
YES


GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs
start sending WI

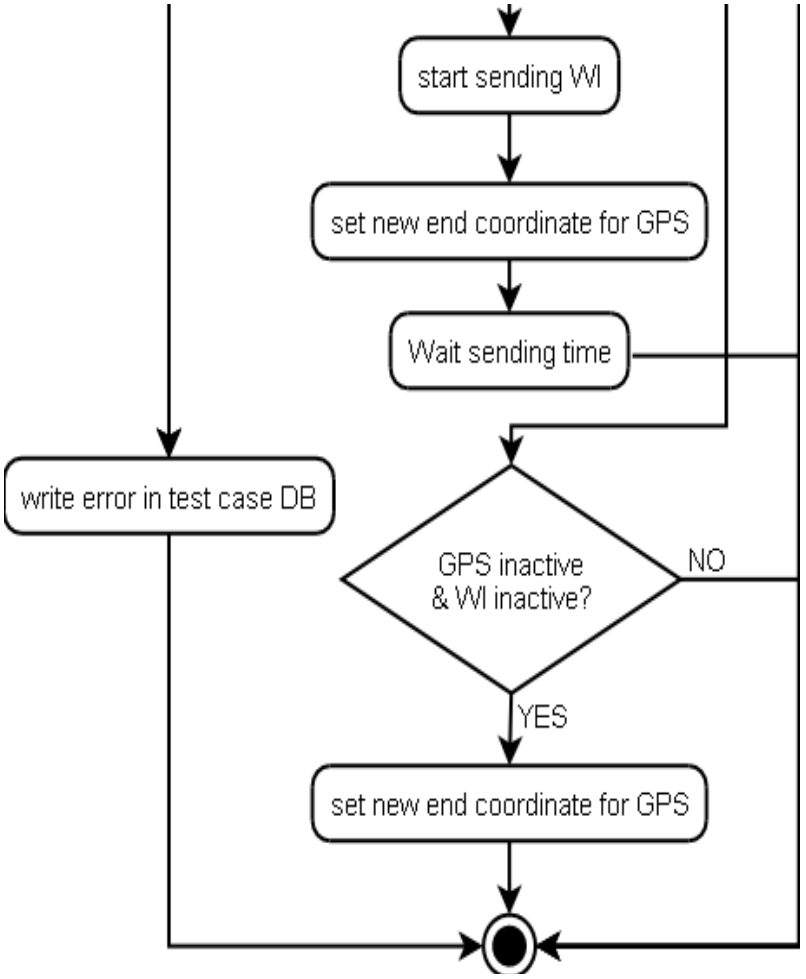
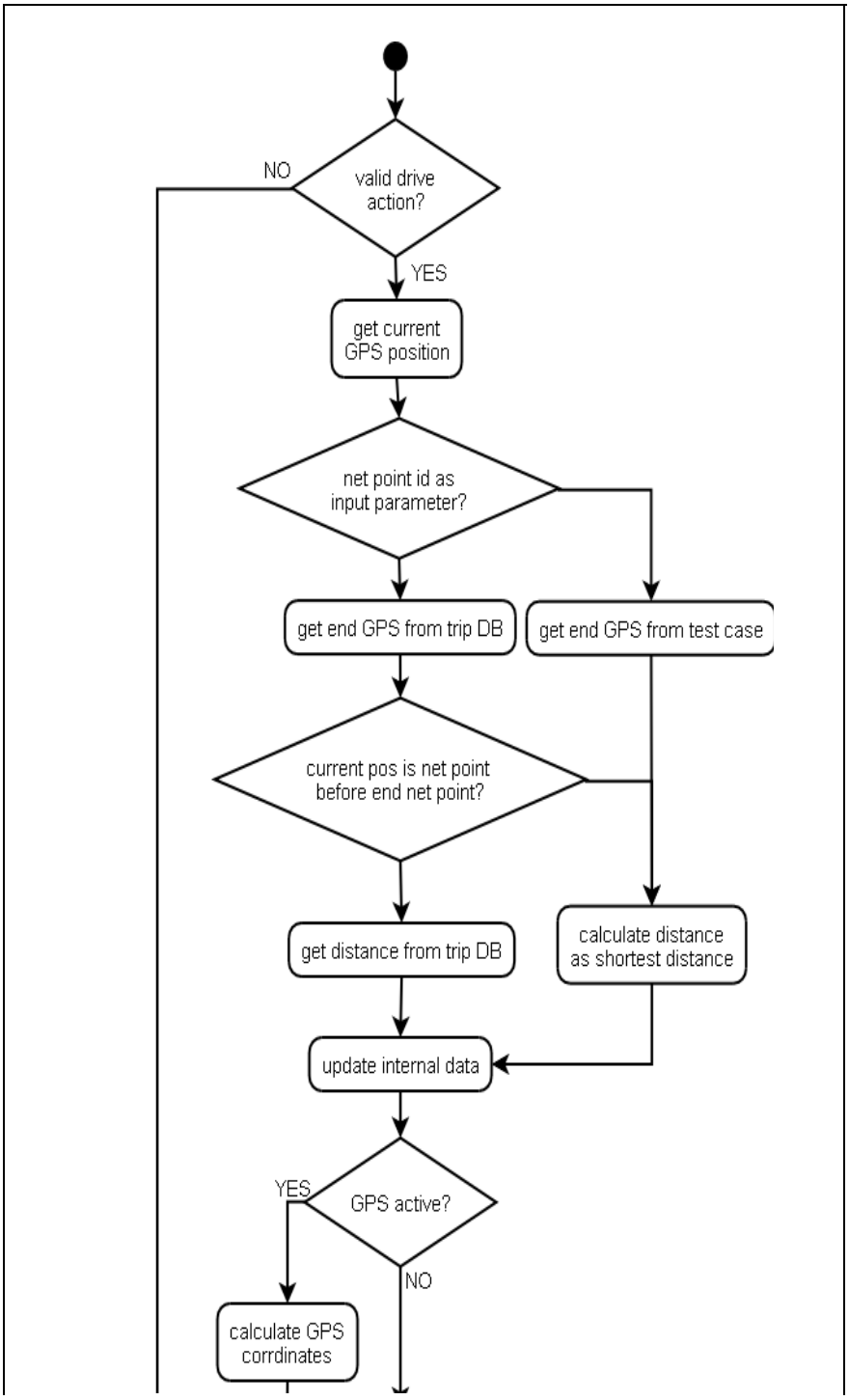
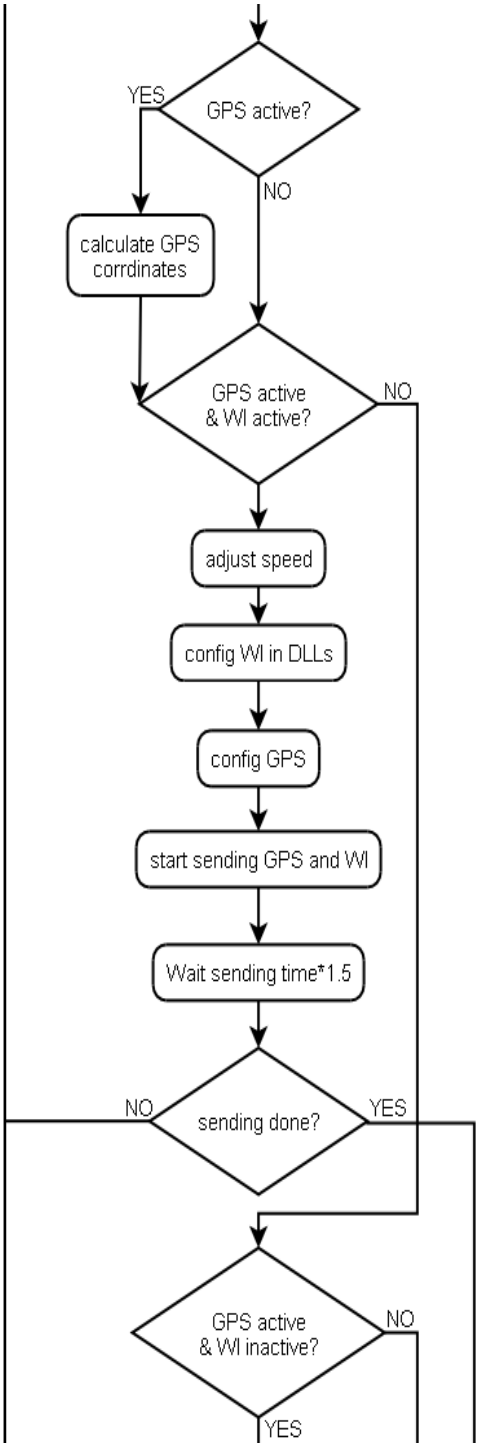
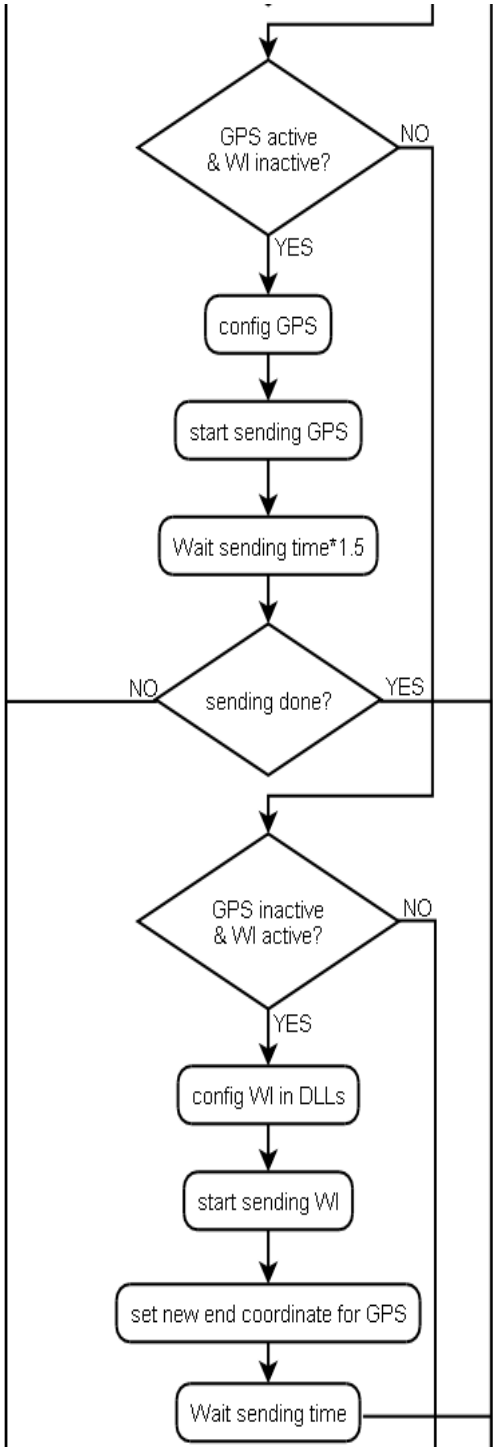


Figure A.52: Activity diagram for the action drive a given distance.


valid drive action?
NO
YES
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
current pos is net point before end net point?
get distance from trip DB
calculate distance as shortest distance
update internal data
GPS active?
YES
NO
calculate GPS corrdinates


GPS active?
YES
NO
calculate GPS corrdinates
GPS active & WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI
Wait sending time*1.5
sending done?
NO
YES
GPS active & WI inactive?
NO
YES


GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs
start sending WI
set new end coordinate for GPS
Wait sending time

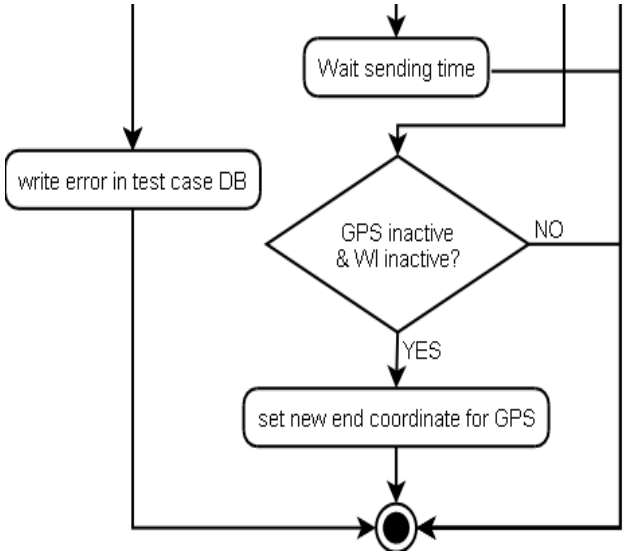
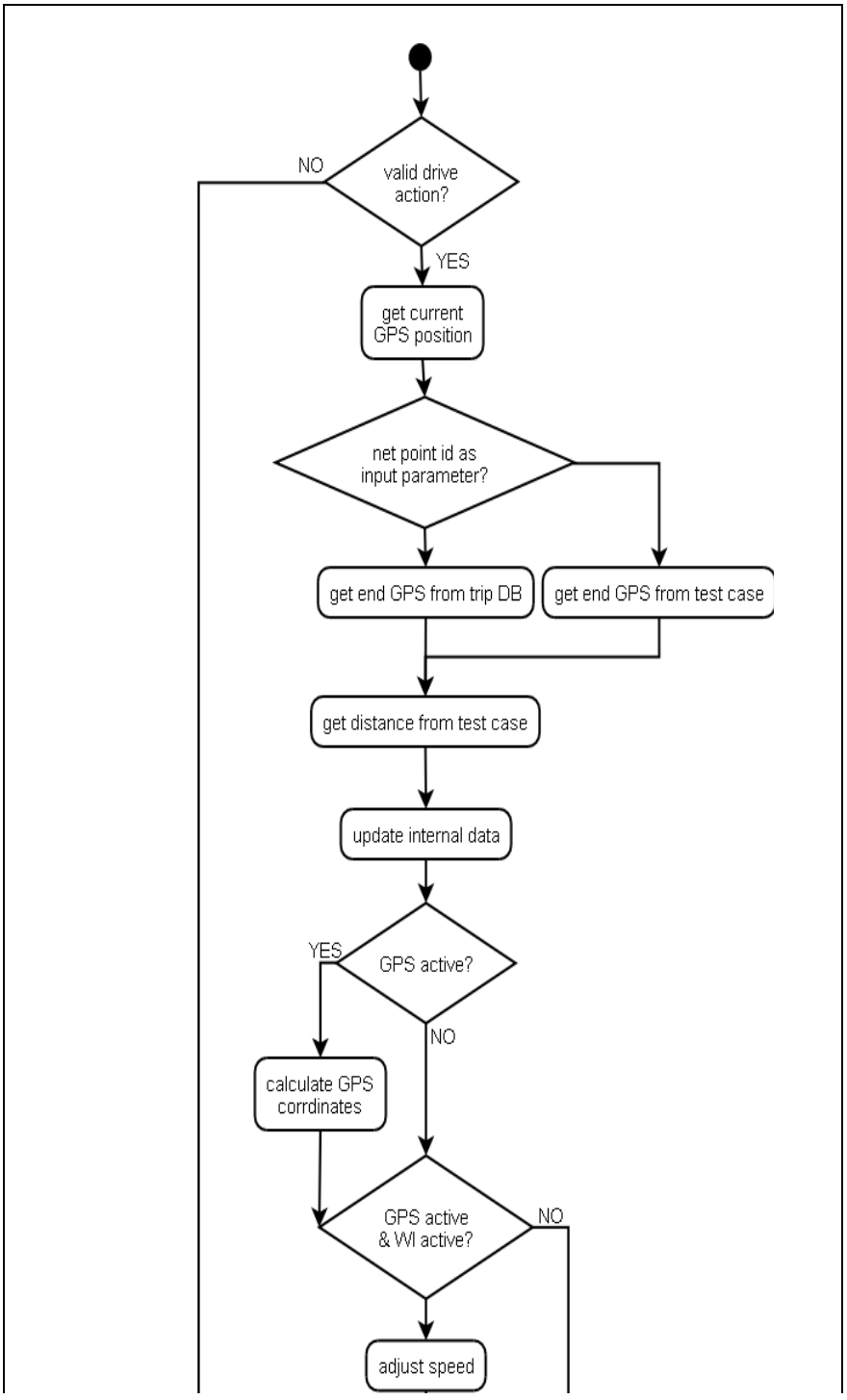
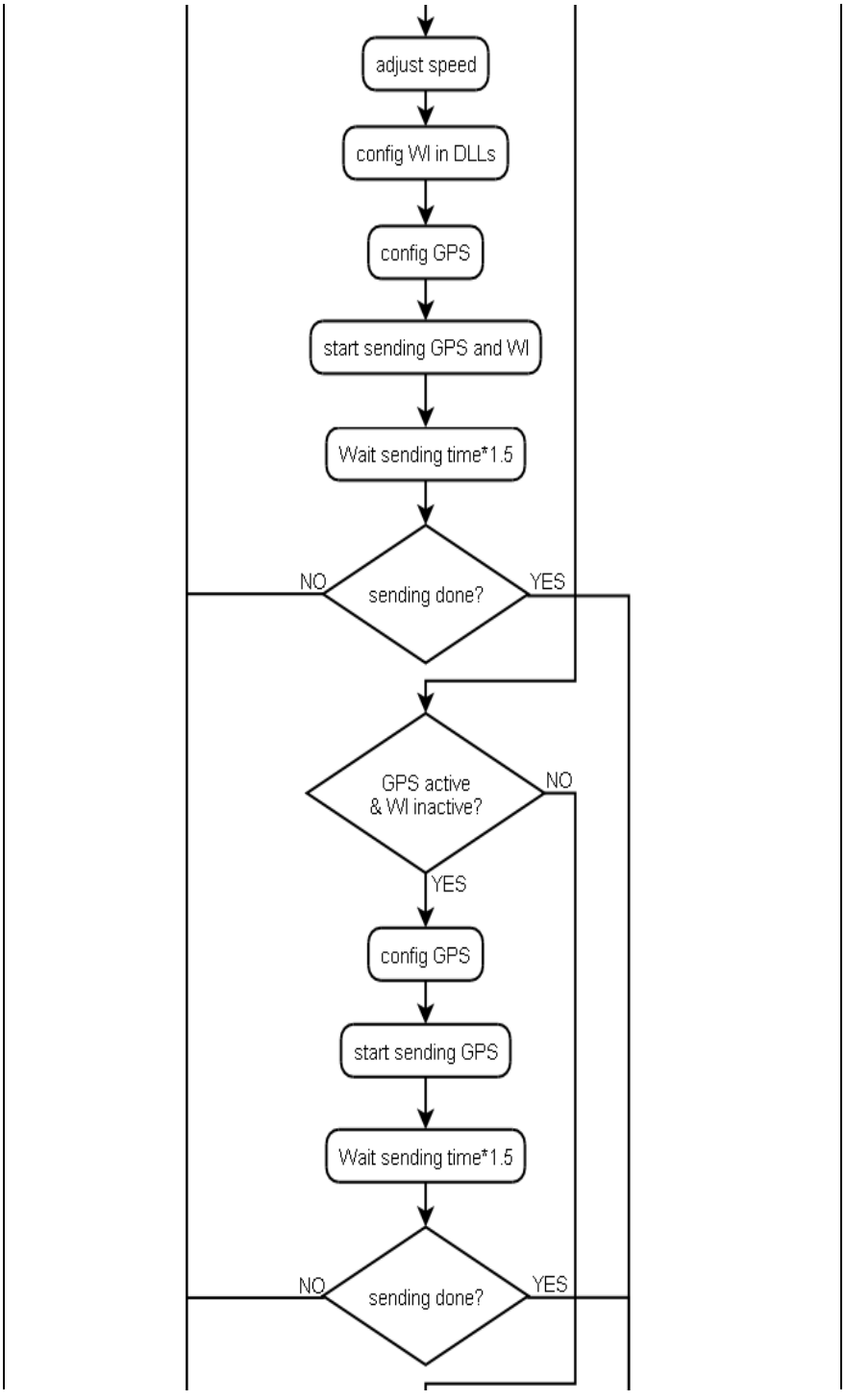


Figure A.53: Activity diagram for the action drive to a specific position.


valid drive action?
NO
YES
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
get distance from test case
update internal data
GPS active?
YES
NO
calculate GPS corrdinates
GPS active & WI active?
NO
adjust speed


adjust speed
config WI in DLLs
config GPS
start sending GPS and WI
Wait sending time*1.5
sending done?
NO
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
sending done?
NO
YES

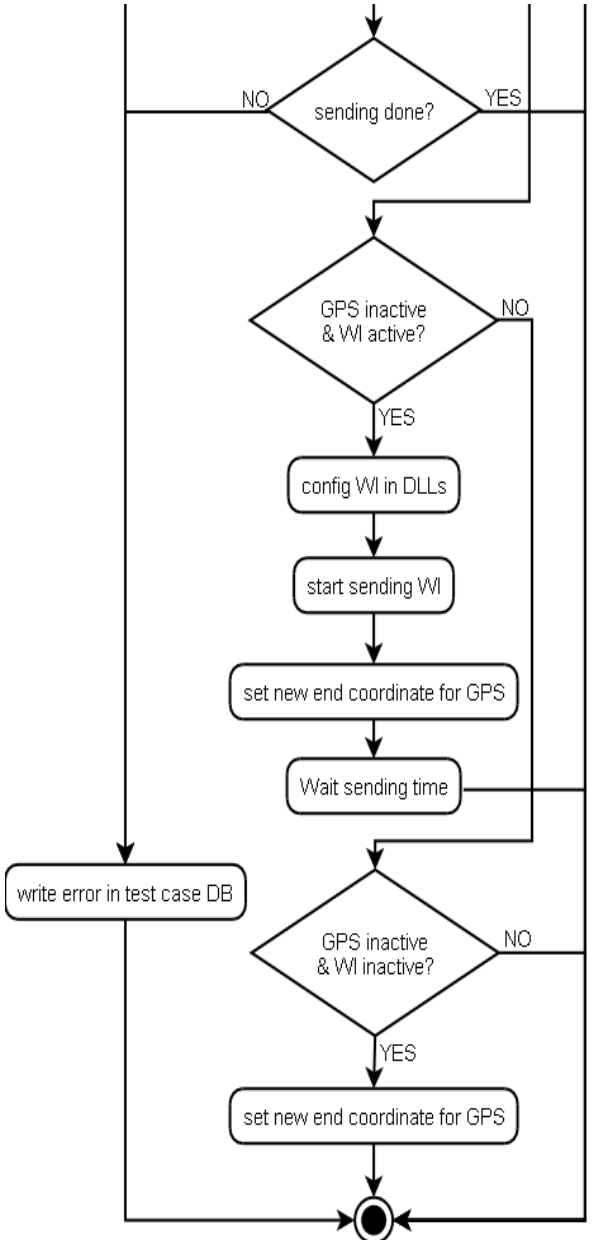
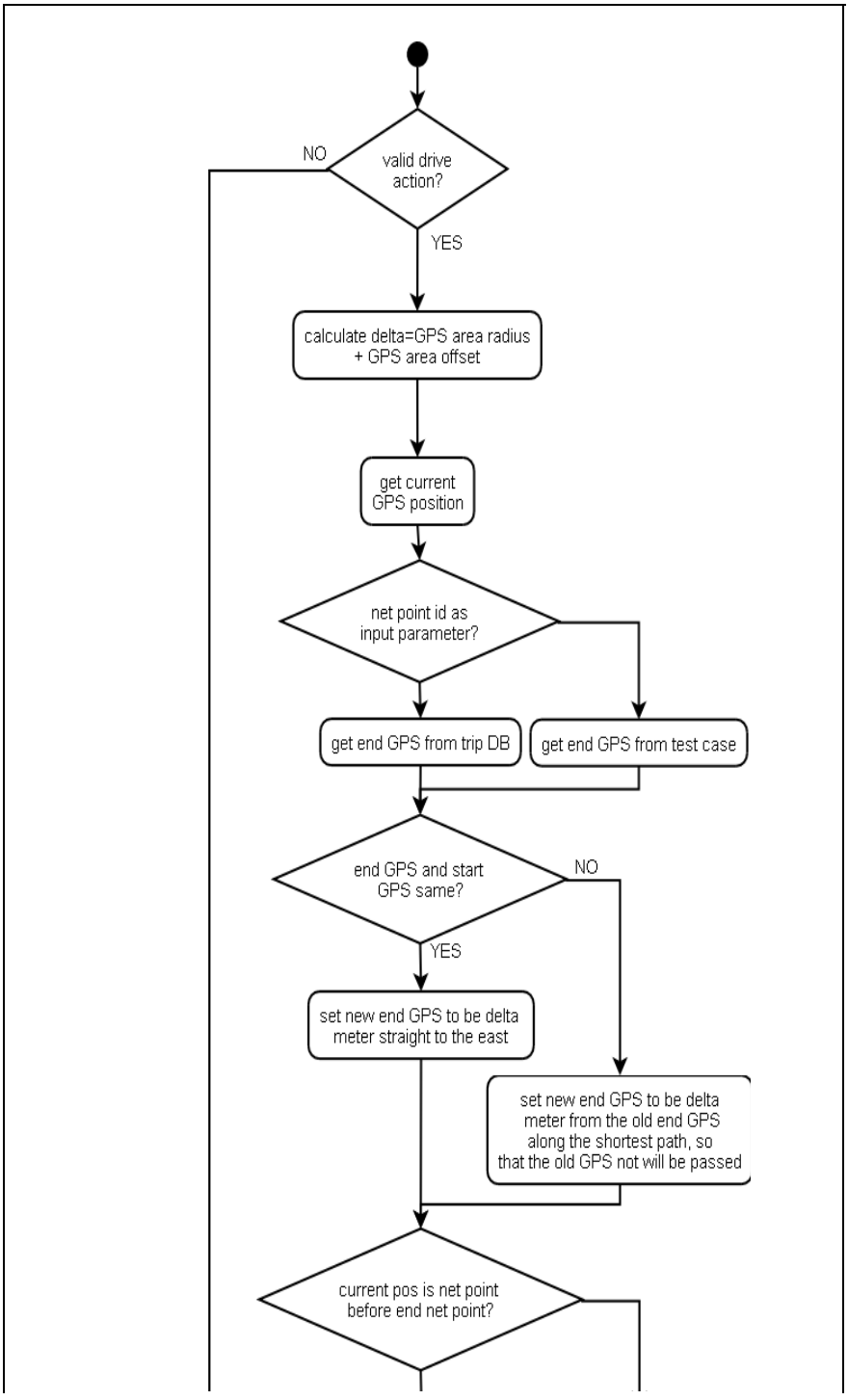
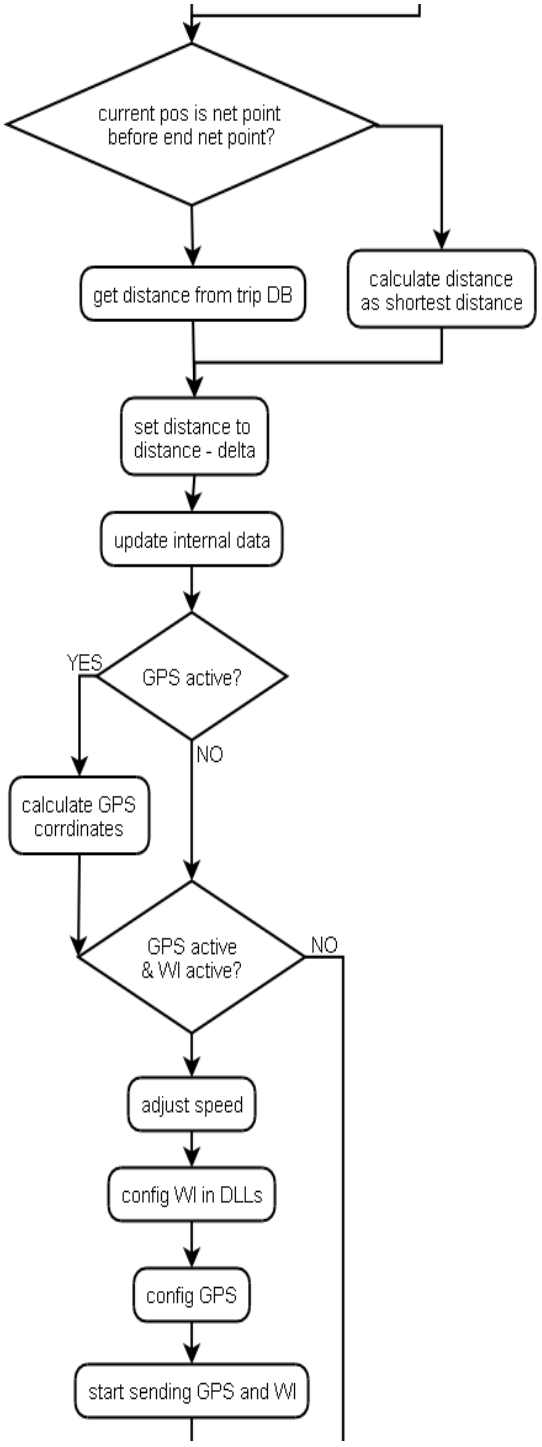
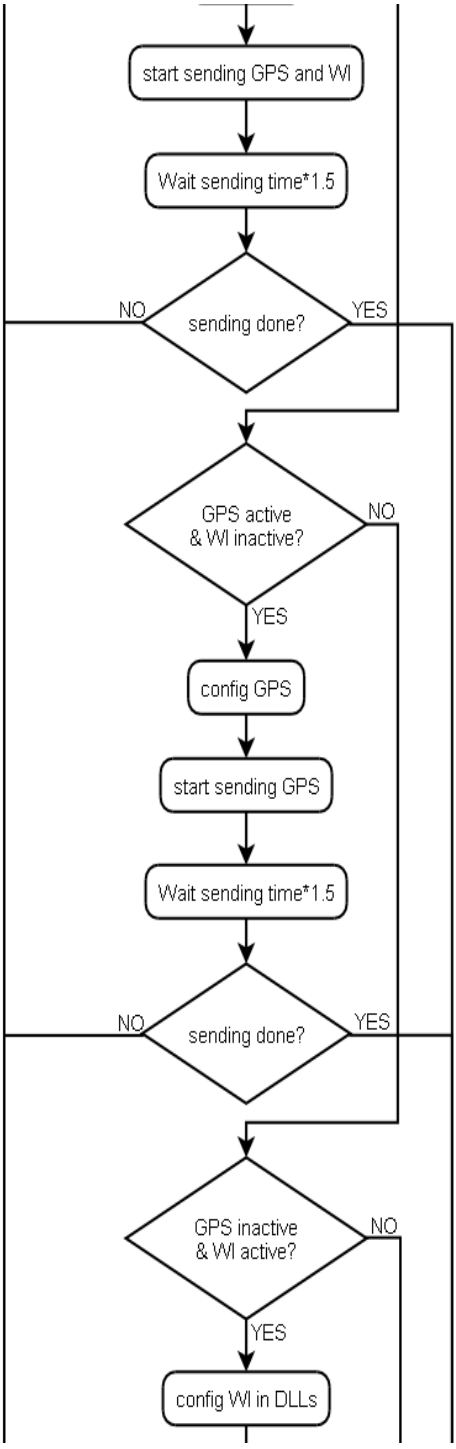


Figure A.54: Activity diagram for the action drive a given distance to a specific position.


valid drive action?
NO
YES
calculate delta=GPS area radius + GPS area offset
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
end GPS and start GPS same?
NO
YES
set new end GPS to be delta meter straight to the east
set new end GPS to be delta meter from the old end GPS along the shortest path, so that the old GPS not will be passed
current pos is net point before end net point?


current pos is net point
before end net point?
get distance from trip DB
calculate distance
as shortest distance
set distance to
distance - delta
update internal data
GPS active?
YES
NO
calculate GPS
corrdinates
GPS active
& WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI


start sending GPS and WI
Wait sending time*1.5
NO
sending done?
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs

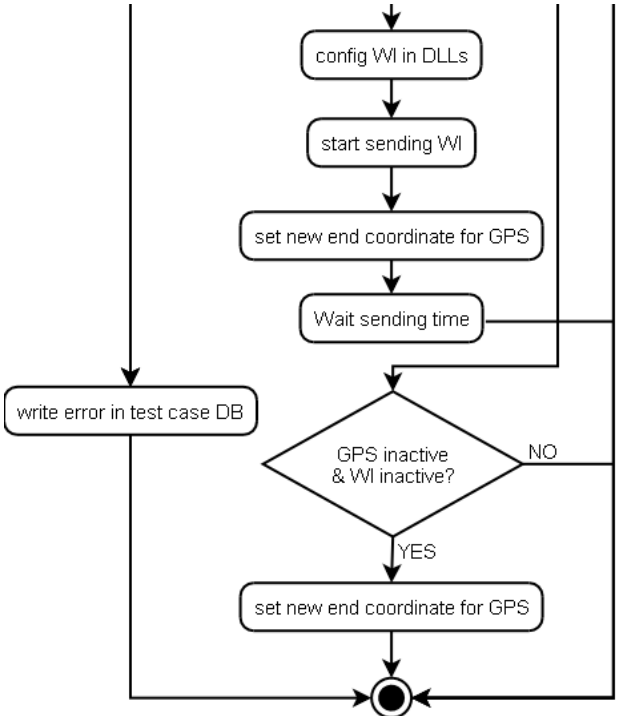
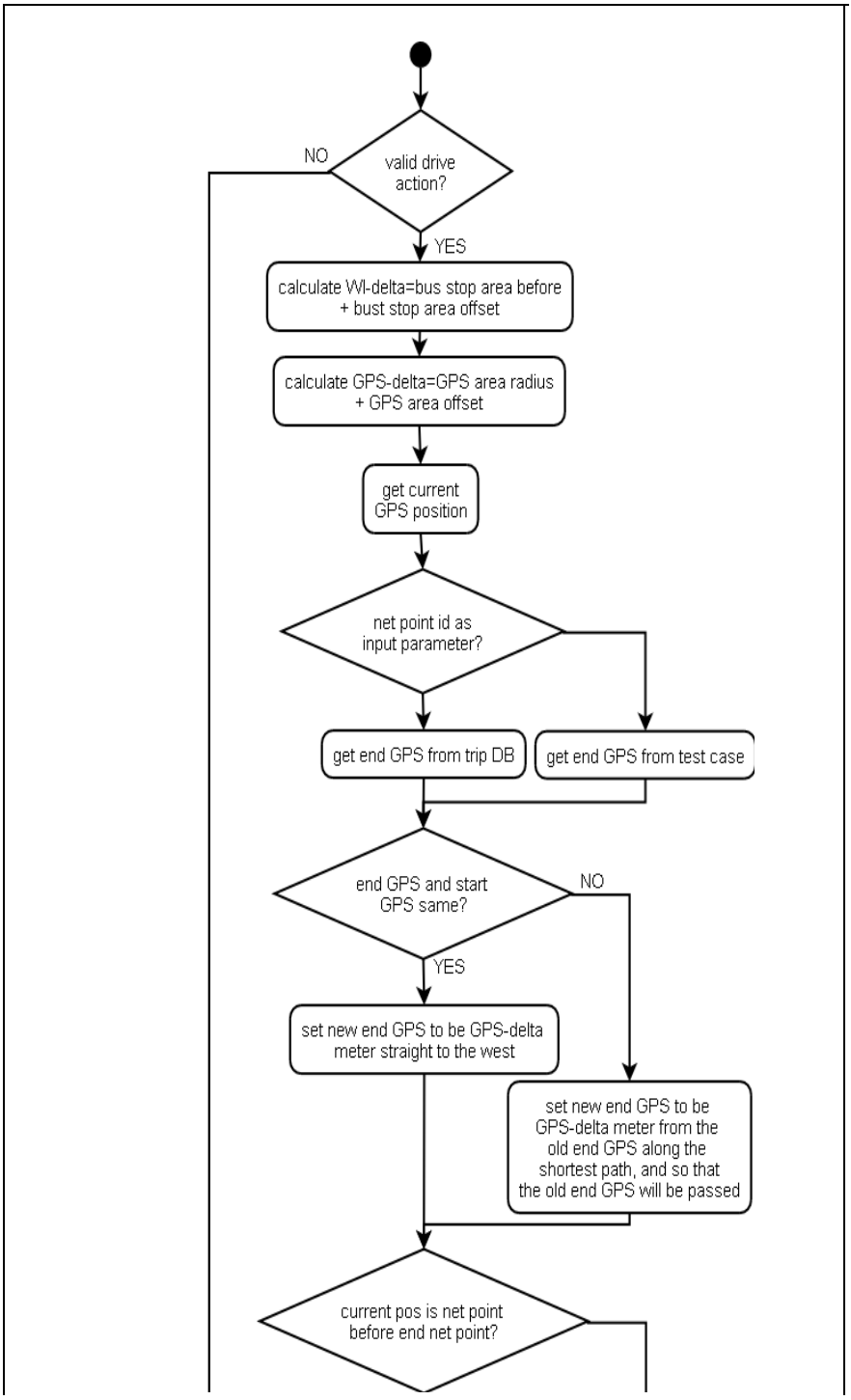
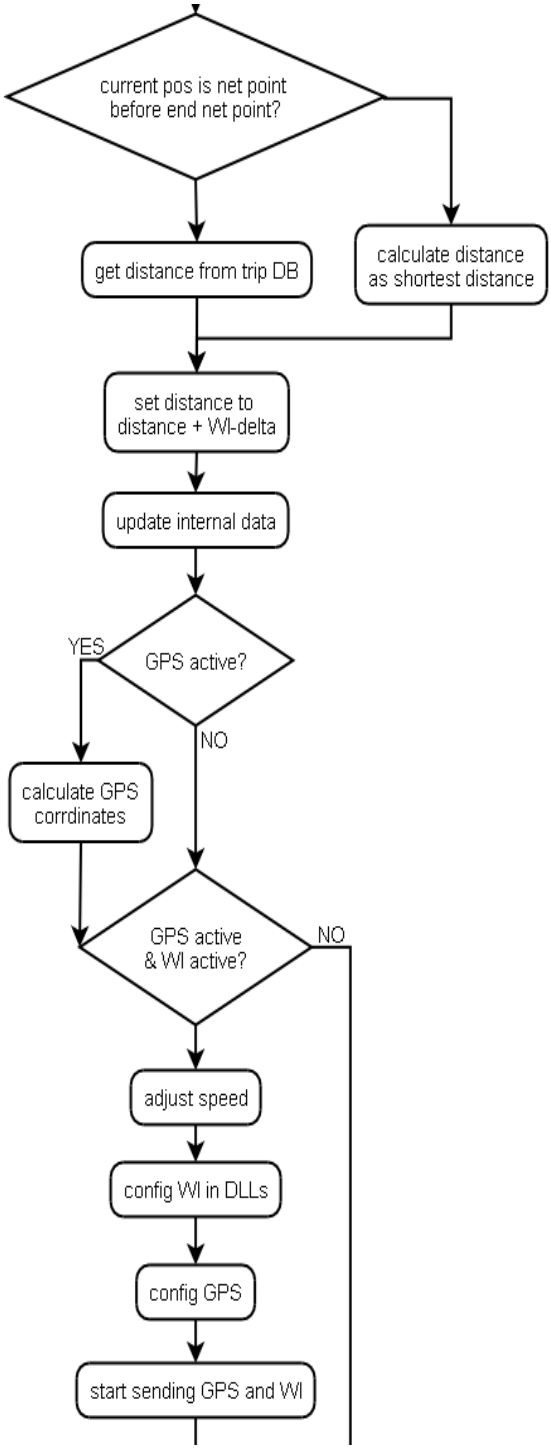
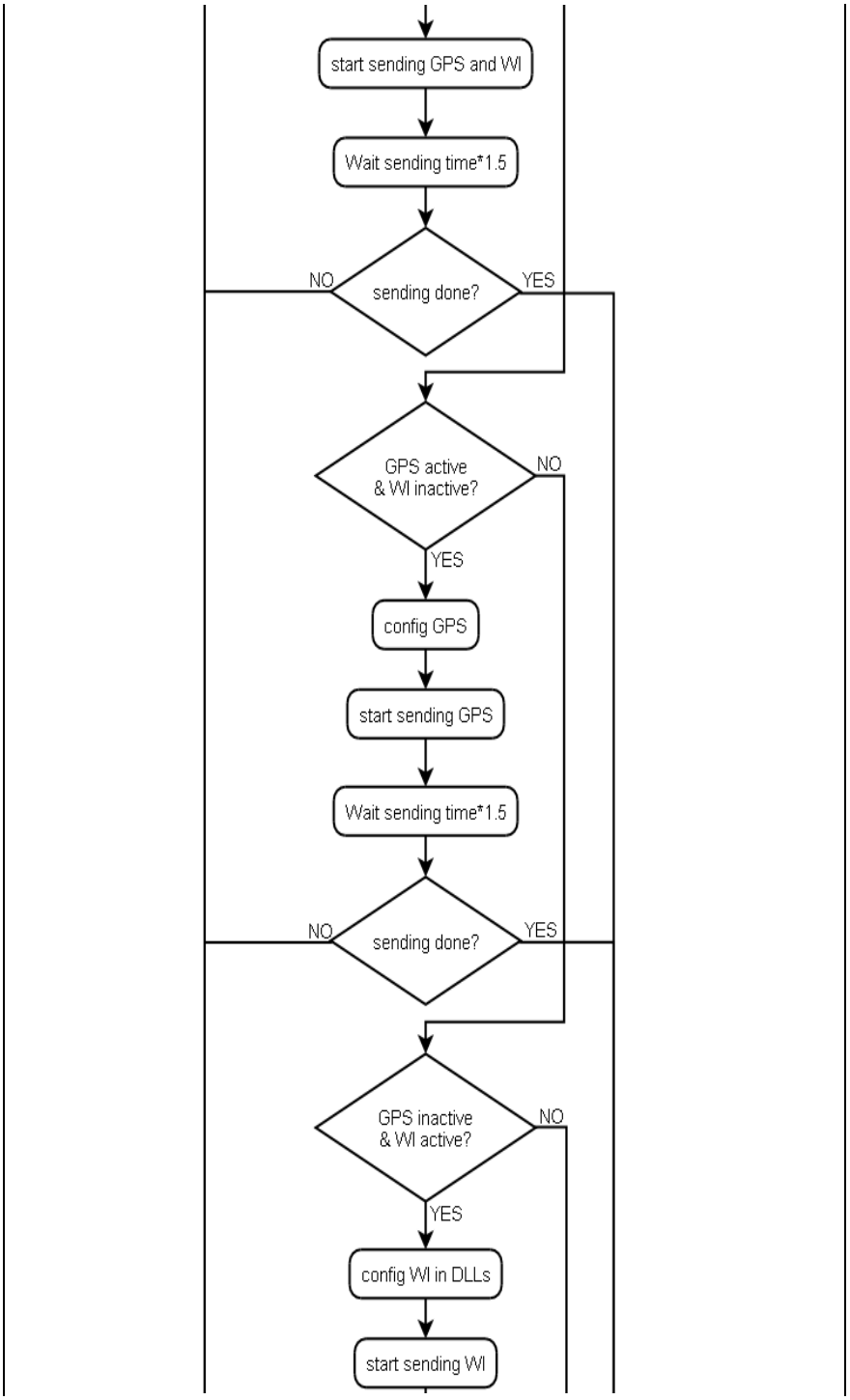


Figure A.55: Activity diagram for the action drive to a point based on the GPS area.


valid drive action?
NO
YES
calculate WI-delta=bus stop area before + bust stop area offset
calculate GPS-delta=GPS area radius + GPS area offset
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
end GPS and start GPS same?
NO
YES
set new end GPS to be GPS-delta meter straight to the west
set new end GPS to be GPS-delta meter from the old end GPS along the shortest path, and so that the old end GPS will be passed
current pos is net point before end net point?


current pos is net point before end net point?
get distance from trip DB
calculate distance as shortest distance
set distance to distance + WI-delta
update internal data
YES
GPS active?
NO
calculate GPS corrdinates
GPS active & WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI


start sending GPS and WI
Wait sending time*1.5
NO
sending done?
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs
start sending WI

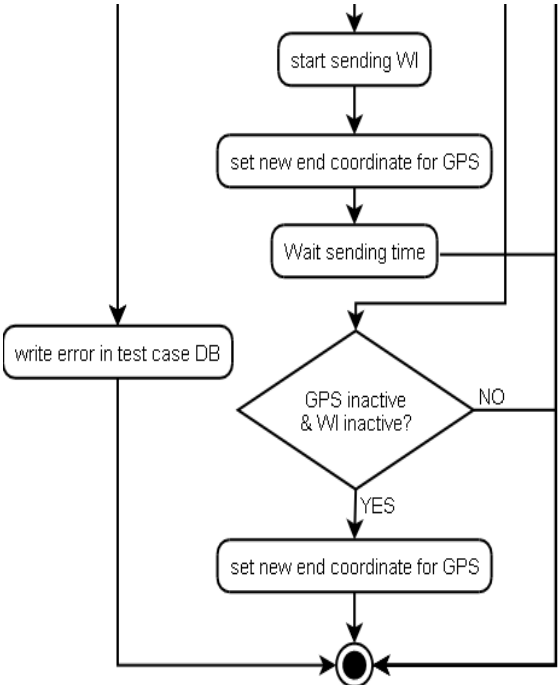
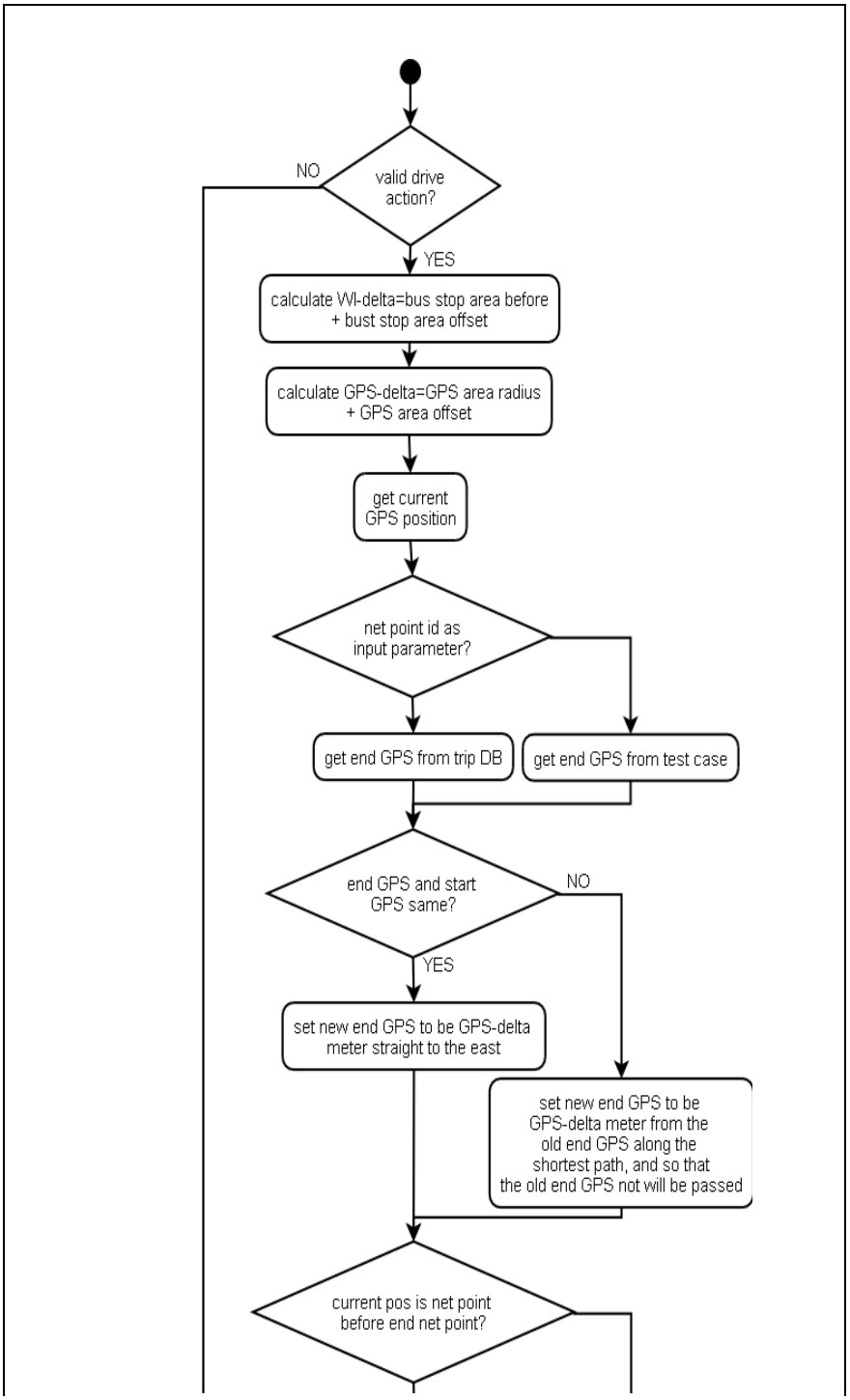
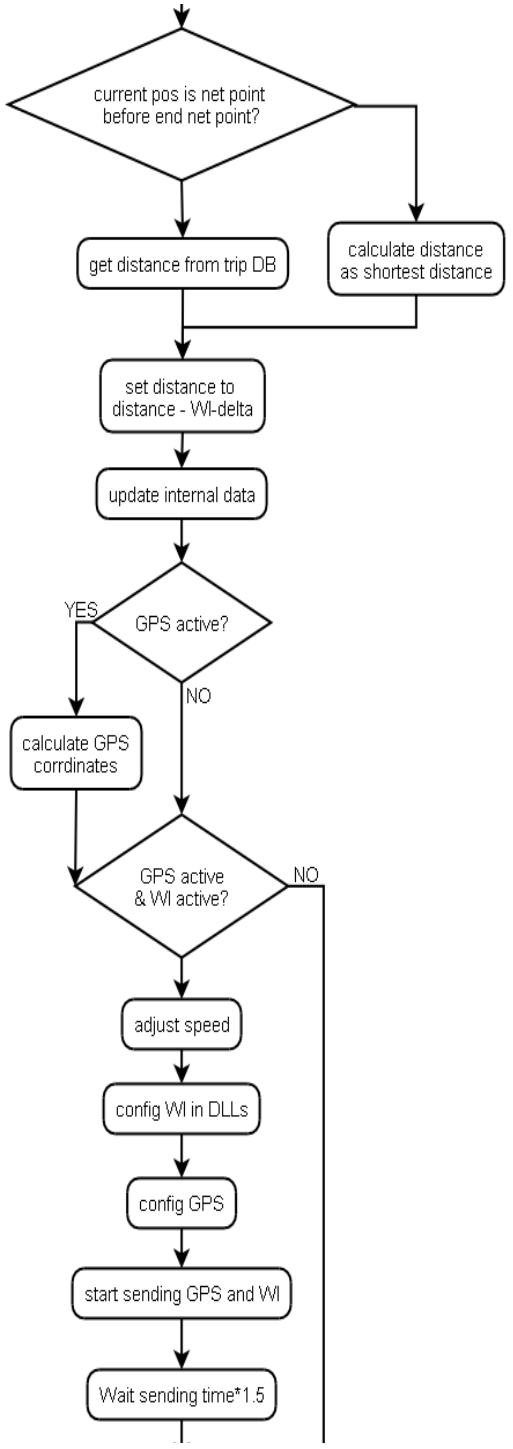
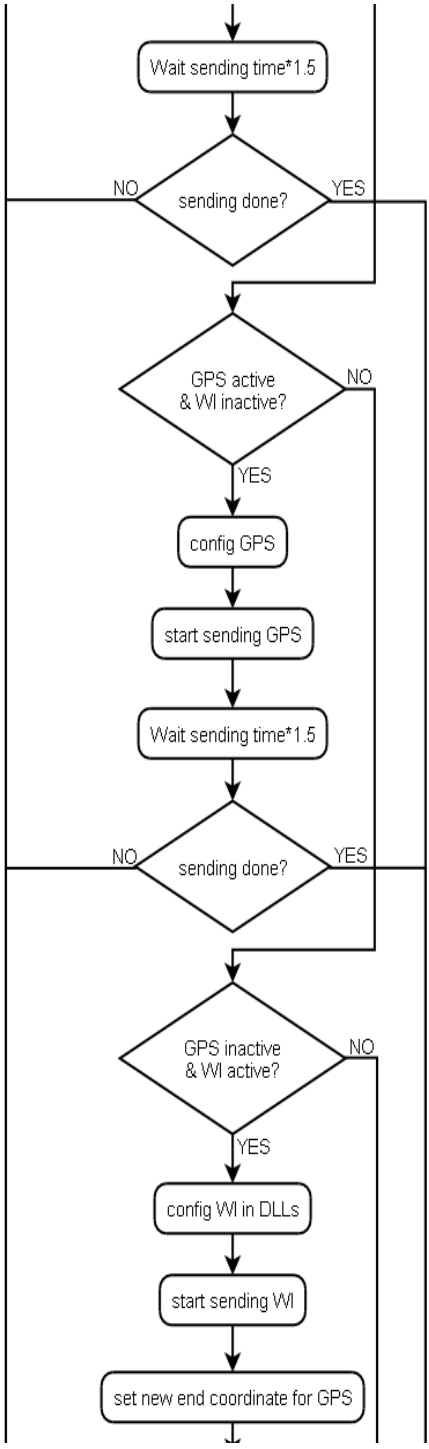


Figure A.56: Activity diagram for the action drive to a point based on stop point area after and the GPS area.


valid drive action?
NO
YES
calculate WI-delta=bus stop area before + bust stop area offset
calculate GPS-delta=GPS area radius + GPS area offset
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
end GPS and start GPS same?
NO
YES
set new end GPS to be GPS-delta meter straight to the east
set new end GPS to be GPS-delta meter from the old end GPS along the shortest path, and so that the old end GPS not will be passed
current pos is net point before end net point?


current pos is net point
before end net point?
get distance from trip DB
calculate distance
as shortest distance
set distance to
distance - WI-delta
update internal data
YES
GPS active?
NO
calculate GPS
corrdinates
GPS active
& WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI
Wait sending time*1.5


Wait sending time*1.5
sending done?
NO
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
sending done?
NO
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs
start sending WI
set new end coordinate for GPS

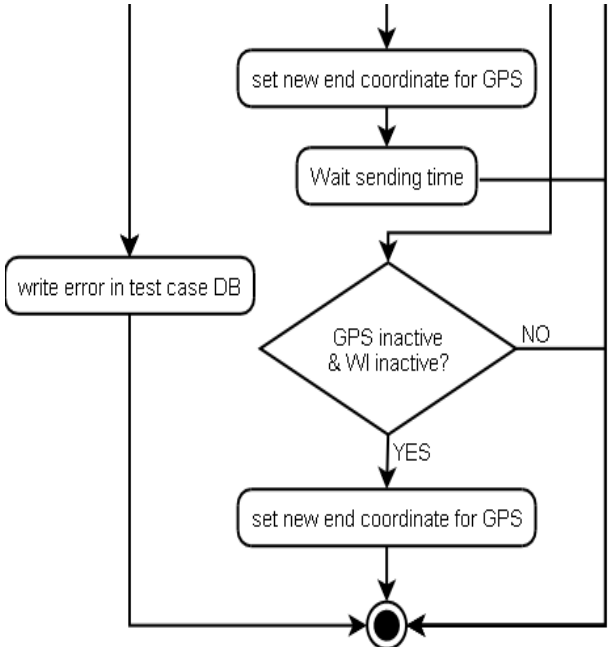
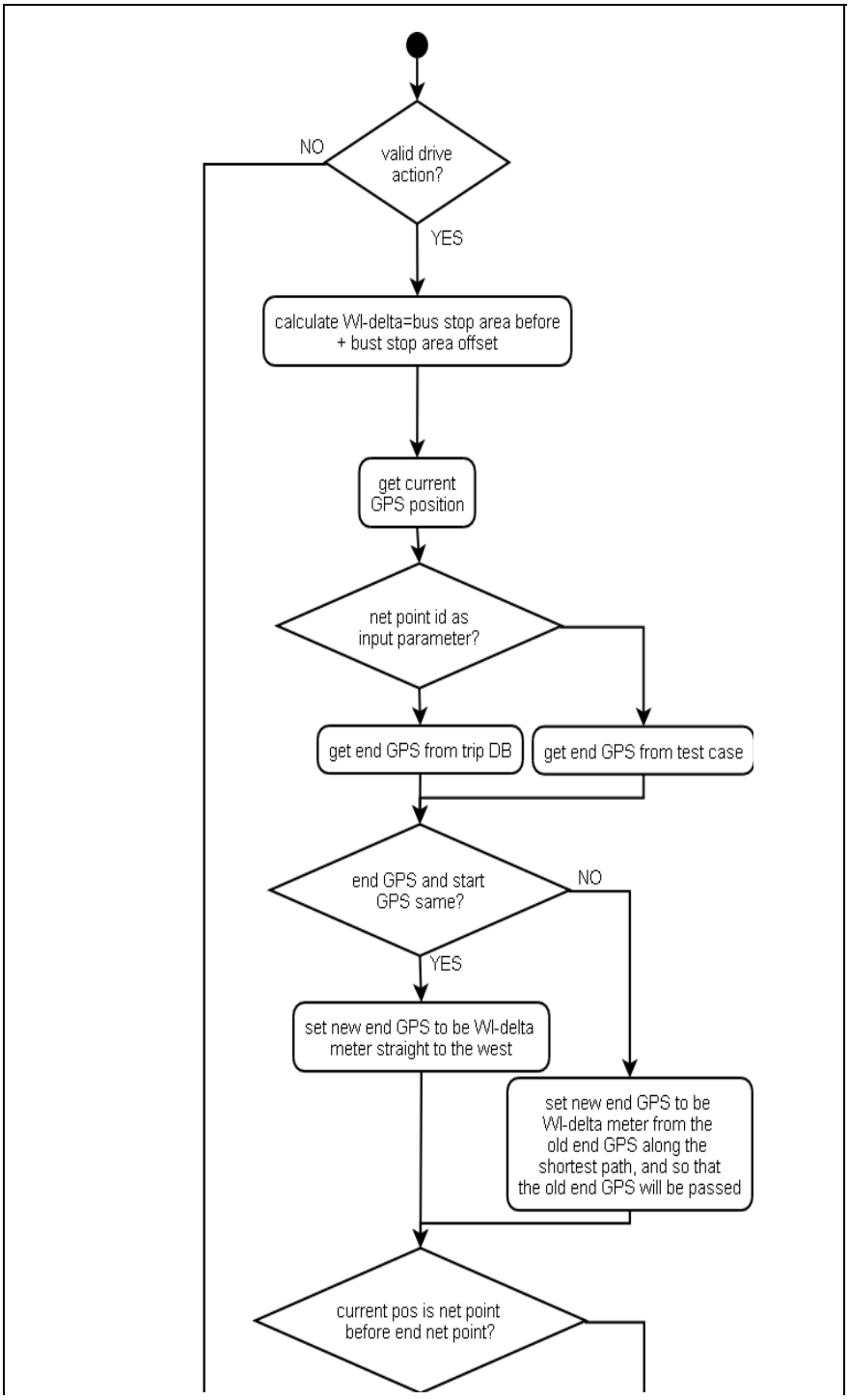
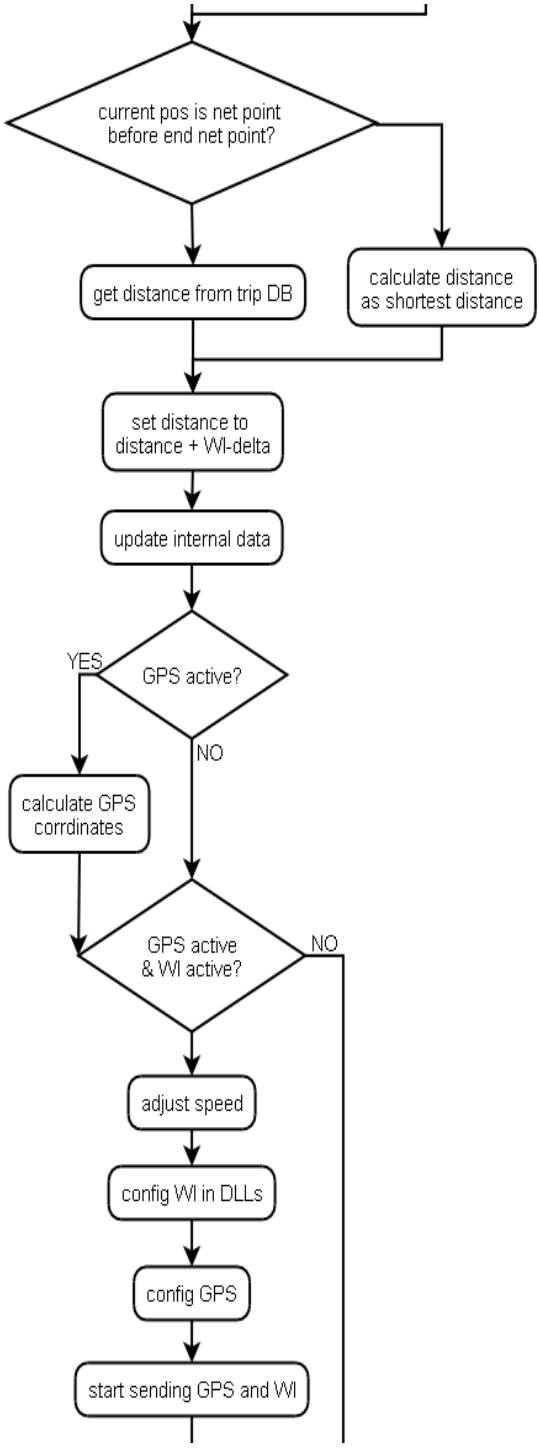
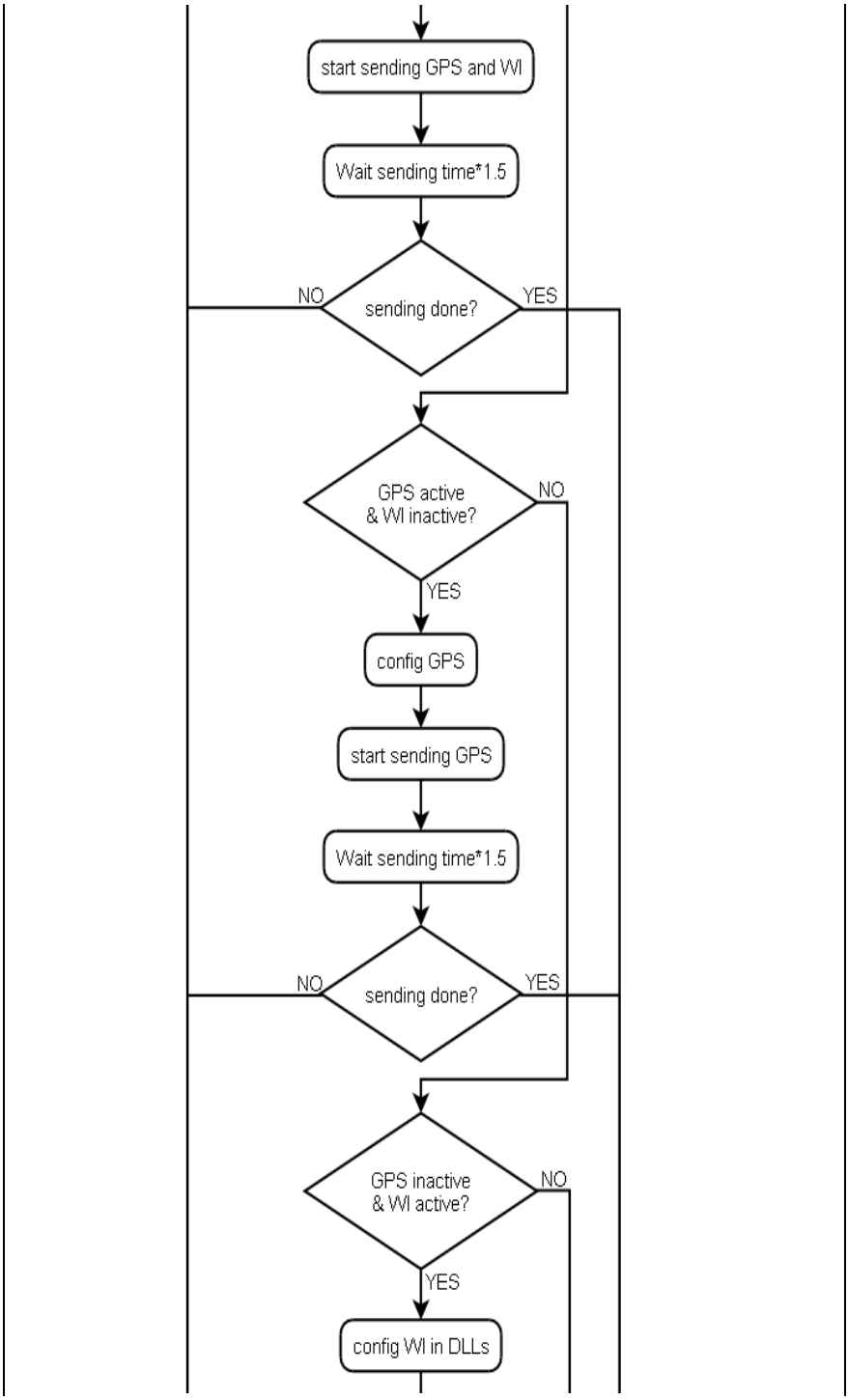


Figure A.57: Activity diagram for the action drive to a point based on stop point area before and the GPS area.


valid drive action?
NO
YES
calculate WI-delta=bus stop area before + bust stop area offset
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
end GPS and start GPS same?
NO
YES
set new end GPS to be WI-delta meter straight to the west
set new end GPS to be WI-delta meter from the old end GPS along the shortest path, and so that the old end GPS will be passed
current pos is net point before end net point?


current pos is net point
before end net point?
get distance from trip DB
calculate distance
as shortest distance
set distance to
distance + WI-delta
update internal data
GPS active?
YES
NO
calculate GPS
corrdinates
GPS active
& WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI


start sending GPS and WI
Wait sending time*1.5
NO
sending done?
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs

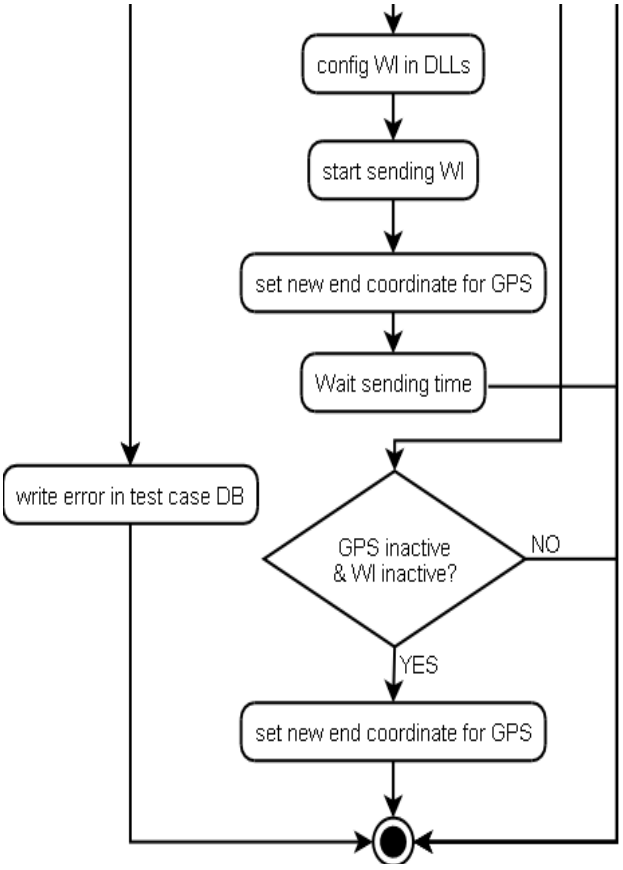
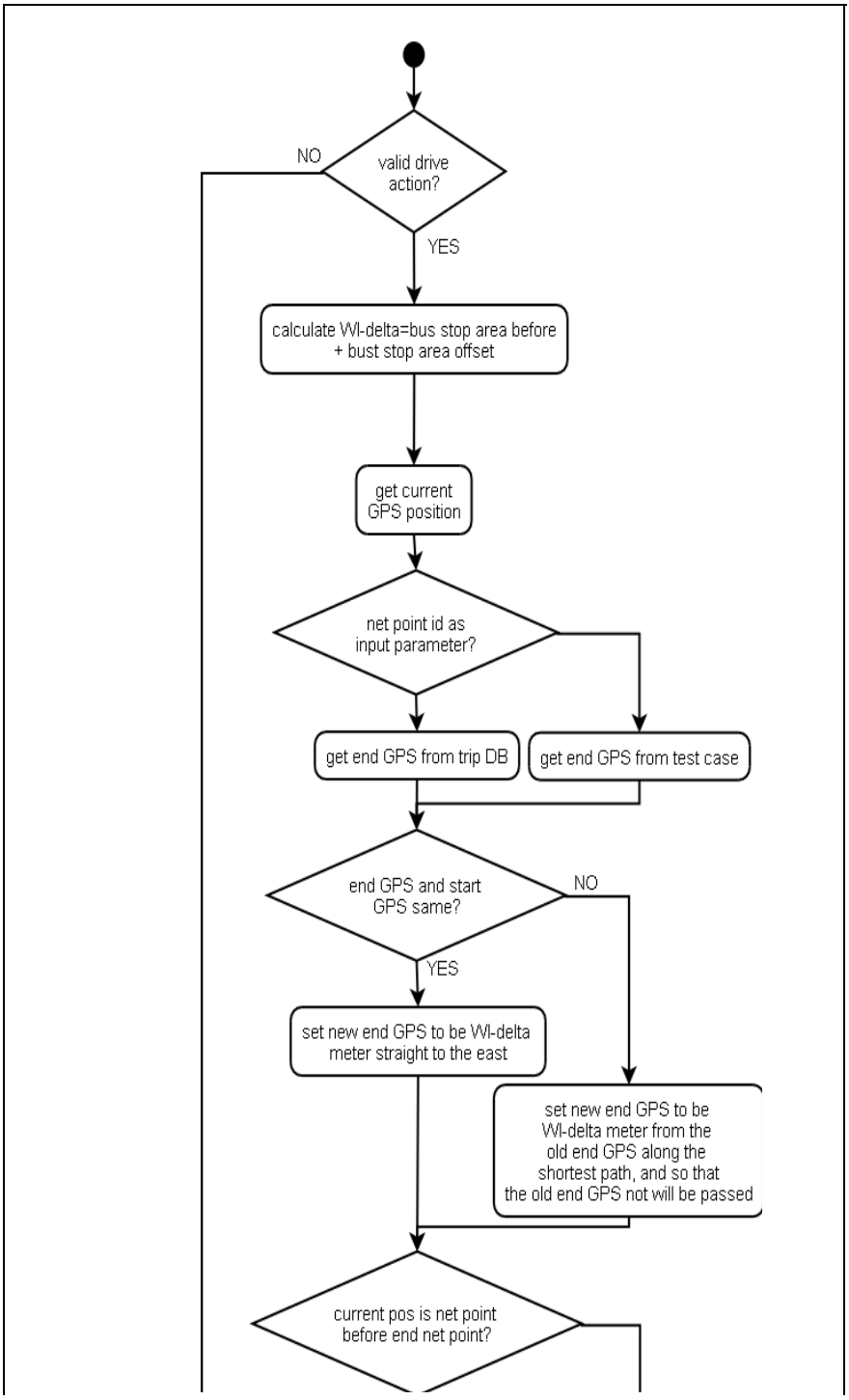
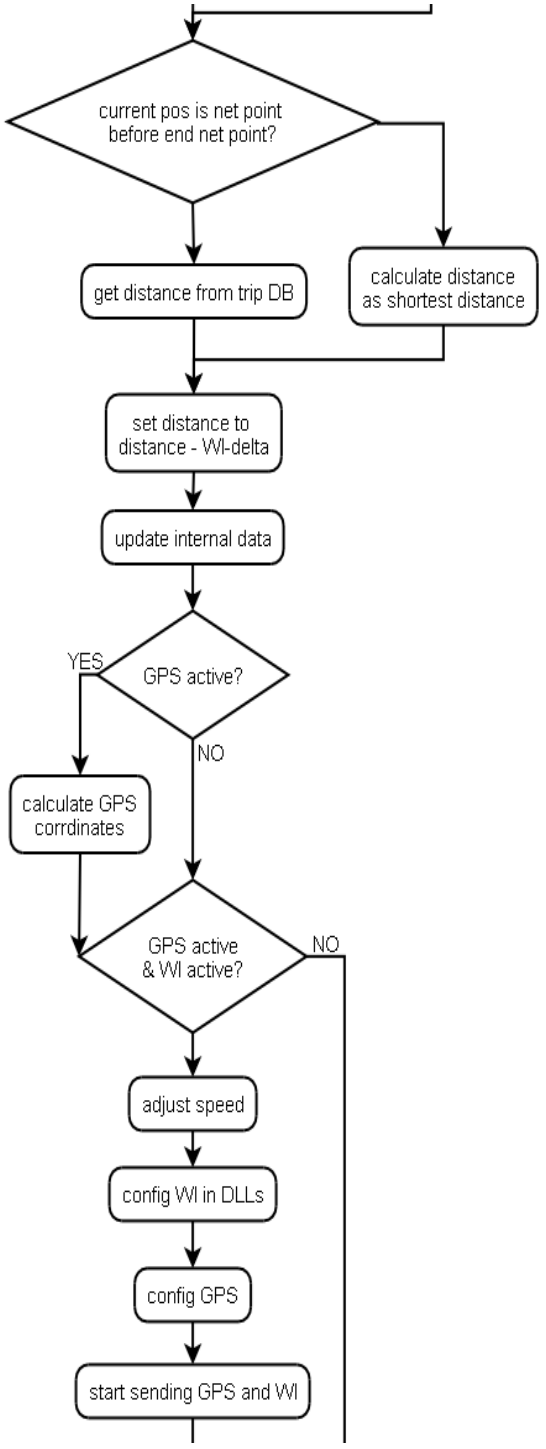
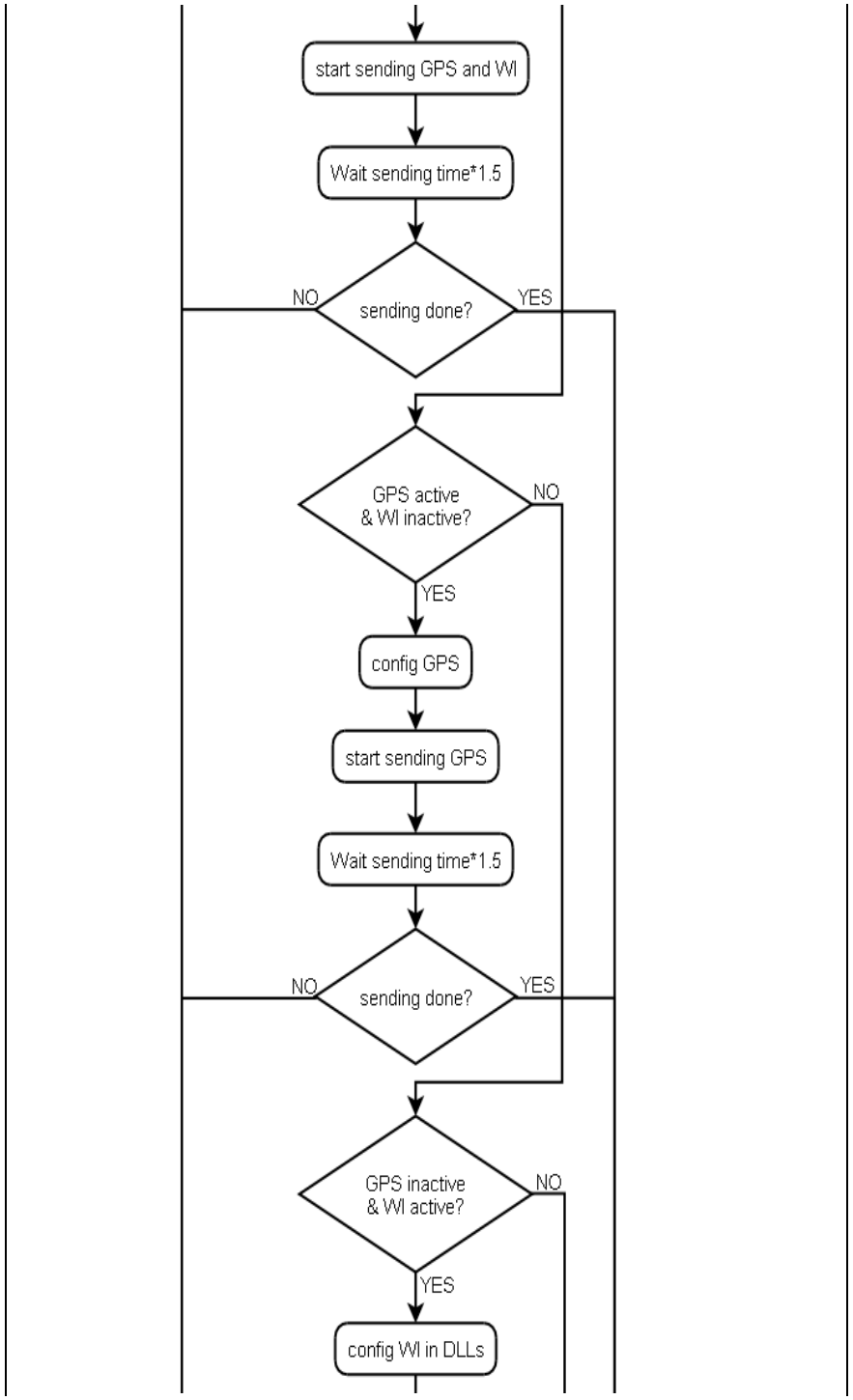


Figure A.58: Activity diagram for the action drive to a point based on stop point area after.


valid drive action?
NO
YES
calculate WI-delta=bus stop area before + bust stop area offset
get current GPS position
net point id as input parameter?
get end GPS from trip DB
get end GPS from test case
end GPS and start GPS same?
NO
YES
set new end GPS to be WI-delta meter straight to the east
set new end GPS to be WI-delta meter from the old end GPS along the shortest path, and so that the old end GPS not will be passed
current pos is net point before end net point?


current pos is net point
before end net point?
get distance from trip DB
calculate distance
as shortest distance
set distance to
distance - WI-delta
update internal data
YES
GPS active?
NO
calculate GPS
corrdinates
GPS active
& WI active?
NO
adjust speed
config WI in DLLs
config GPS
start sending GPS and WI


start sending GPS and WI
Wait sending time*1.5
NO
sending done?
YES
GPS active
& WI inactive?
NO
YES
config GPS
start sending GPS
Wait sending time*1.5
NO
sending done?
YES
GPS inactive
& WI active?
NO
YES
config WI in DLLs

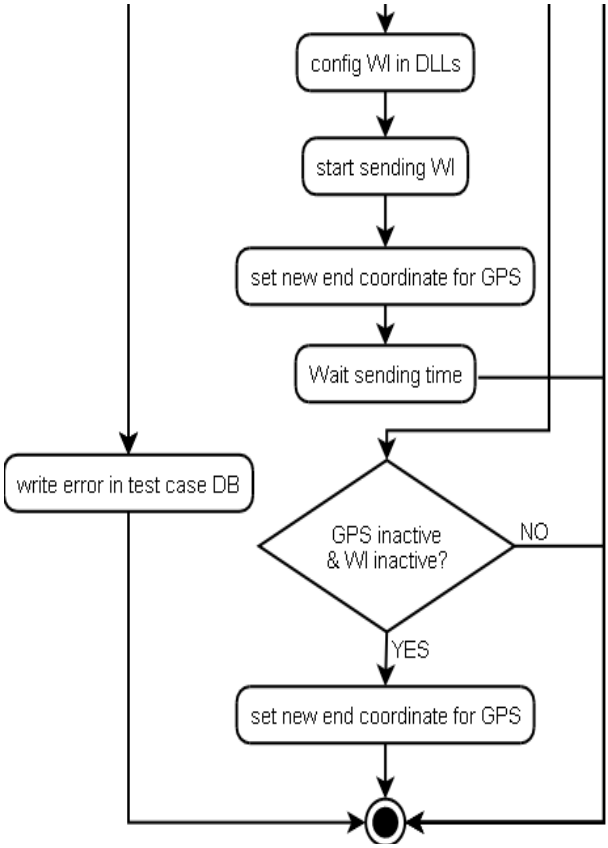


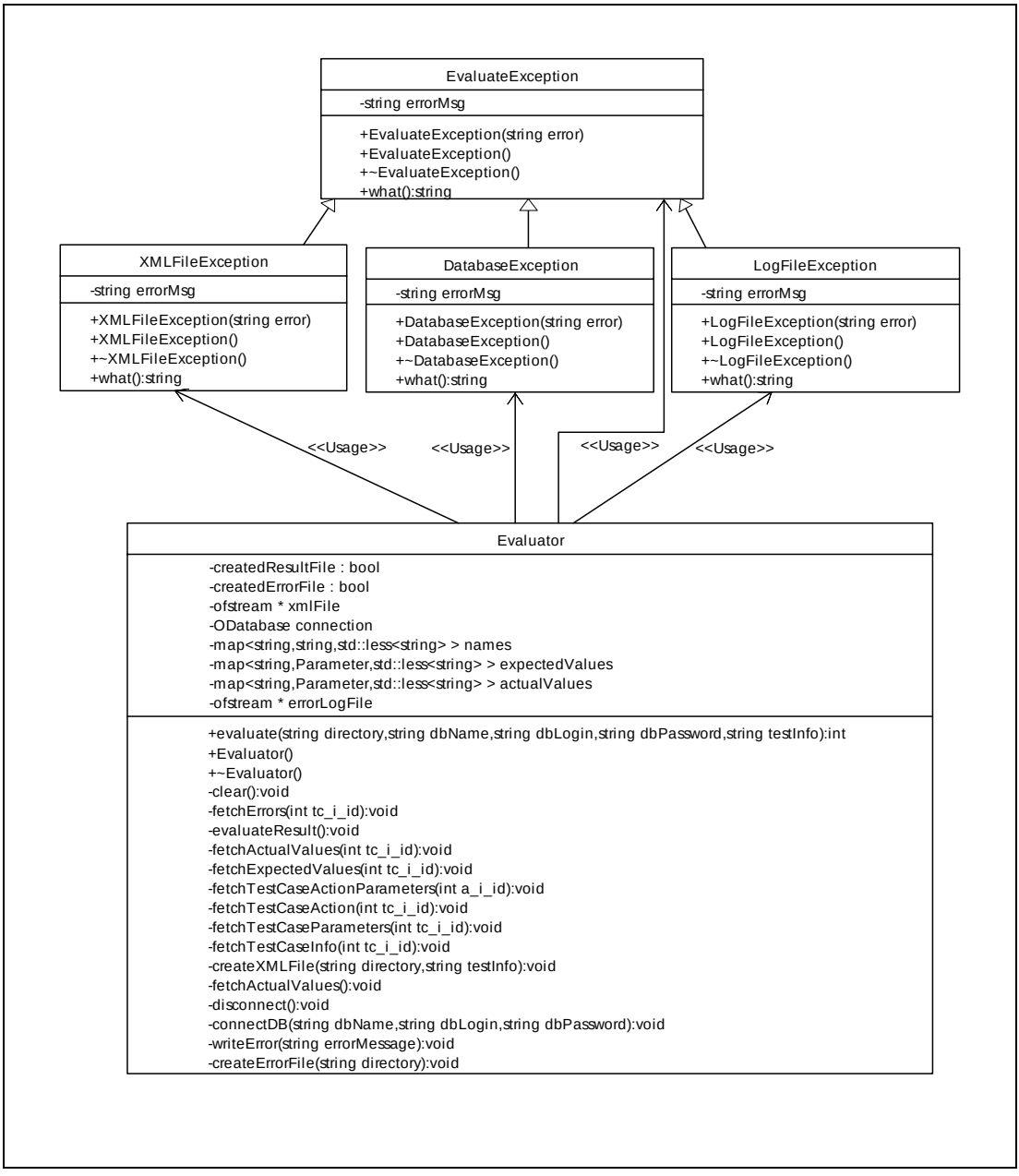
Figure A.59: Activity diagram for the action drive to a point based on stop point area before.

A.2 The test evaluation

A.2.4 Detailed class diagrams

The following figures show the detailed description of the classes including their collaboration with other classes.

Figure A.60: Class diagram for the test evaluation.


EvaluateException
-string errorMsg
+EvaluateException(string error)
+EvaluateException()
+~EvaluateException()
+what():string
XMLFileException
-string errorMsg
+XMLFileException(string error)
+XMLFileException()
+~XMLFileException()
+what():string
DatabaseException
-string errorMsg
+DatabaseException(string error)
+DatabaseException()
+~DatabaseException()
+what():string
LogFileException
-string errorMsg
+LogFileException(string error)
+LogFileException()
+~LogFileException()
+what():string
<<Usage>>
<<Usage>>
<<Usage>>
<<Usage>>
Evaluator
-createdResultFile : bool
-createdErrorFile : bool
-ofstream * xmlFile
-ODatabase connection
-map<string,string,std::less<string> > names
-map<string,Parameter,std::less<string> > expectedValues
-map<string,Parameter,std::less<string> > actualValues
-ofstream * errorLogFile
+evaluate(string directory,string dbName,string dbLogin,string dbPassword,string testInfo):int
+Evaluator()
+~Evaluator()
-clear():void
-fetchErrors(int tc_i_id):void
-evaluateResult():void
-fetchActualValues(int tc_i_id):void
-fetchExpectedValues(int tc_i_id):void
-fetchTestCaseActionParameters(int a_i_id):void
-fetchTestCaseAction(int tc_i_id):void
-fetchTestCaseParameters(int tc_i_id):void
-fetchTestCaseInfo(int tc_i_id):void
-createXMLFile(string directory,string testInfo):void
-fetchActualValues():void
-disconnect():void
-connectDB(string dbName,string dbLogin,string dbPassword):void
-writeError(string errorMessage):void
-createErrorFile(string directory):void

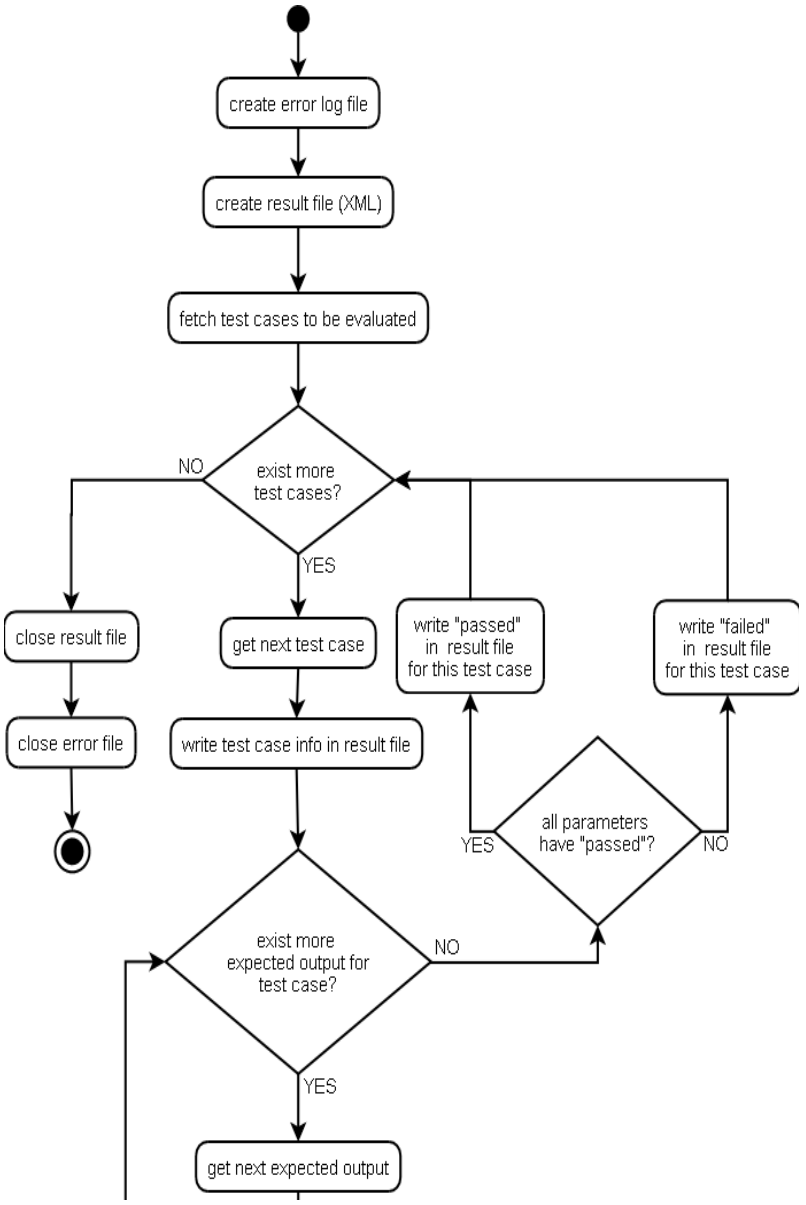
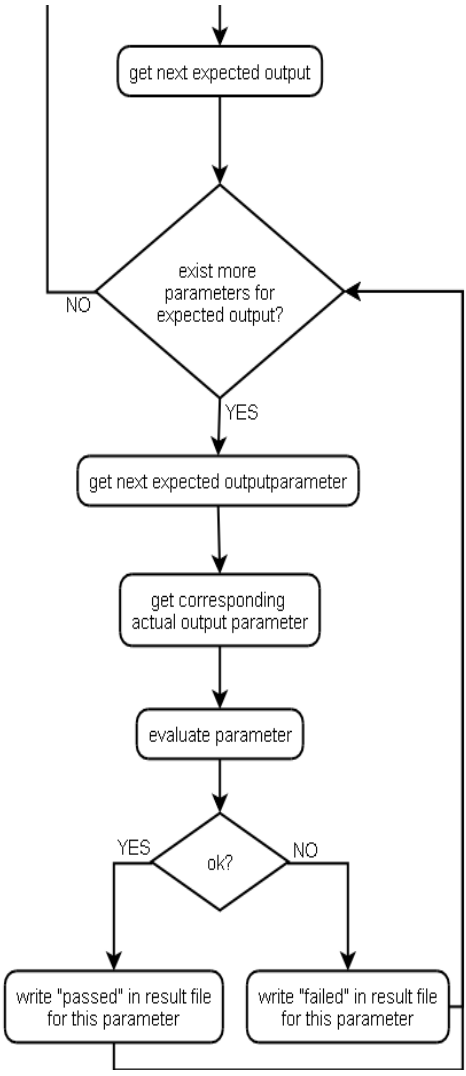


Figure A.61: Activity diagram of the test evaluation.


get next expected output
exist more parameters for expected output?
NO
YES
get next expected outputparameter
get corresponding actual output parameter
evaluate parameter
YES
ok?
NO
write "passed" in result file for this parameter
write "failed" in result file for this parameter

BIBLIOGRAPHY

[Beizer 1990] Beizer, Boris (1990). *Software Testing Techniques.* New York, USA: International Thomson Computer Press.

[Burnstein 2003] Burnstein, Ilene (2003). *Practical Software Testing.* New York, NY, USA: Springer Science+Business Media, Inc.

[Camposano 1996] Camposano, R. & Wilberg, J (1996). "Embedded System Design" in *Design Automation for Embedded Systems*, vol. 1, pp. 5-50.

[Copeland 2004] Copeland, Lee (2004). *A Practitioner's Guide to Software Test Design.* Artech House.

[Fewster 1999] Fewster, Mark & Graham, Dorothy (1999). *Software Test Automation.* Great Britain: Addison Wesley.

[Gamma 1995] Gamma, Erich; Helm, Richard; Johnson, Ralph & Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley.

[Hetzel 1993] Hetzel, Bill (1993). *Making Software Measurement Work.* Wellesley, USA: QED.

[IEEE 1990] (1990). *IEEE Standard Glossary of Software Engineering Terminology* Std 610.12-1990.

[IEEE 1998] (1998). *IEEE Standard for Software Test Documents* Std 829-1998

[Kandler 2000] Kandler, Jim (2000). *Automated Testing of Embedded Software - Lessons learned from a Successful Implementation*.

[Karlsson 2006] Karlsson, Daniel (2006). *Verification of Component-based Embedded System Design.* Linköping, Sweden: Linköping university.

[Loveland 2004] Loveland, Scott (2004). *Software Testing Techniques: Finding the Defects That Matter.* Hingham, MA, USA: Charles River Media.

[Myers 2004] Myers, Glenford J (2004). *Art of Software Testing.* Hoboken, NJ, USA: John Wiley & Sons, Incorporated.

[VöV 1984] Verband öffentlicher Verkehrsbetriebe (1984). *Integriertes Bordinformationssystem (IBIS).* Germany: VDV Schriften, 1984.

[VöV 1987] Verband öffentlicher Verkehrsbetriebe (1987). *Integriertes Bordinformationssystem (IBIS), Änderungen/Ergänzungen zur Ausgabe Juni 1984.* Germany: VDV Schriften.

[VöV 1991] Verband öffentlicher Verkehrsbetriebe (1991). *Integriertes Bordinformationssystem (IBIS),* Ergänzung 2. Germany: VDV Schriften.

[Watkins 2001] Watkins, John (2001). *Testing IT: An Off-the-Shelf Software Testing Handbook.* Port Chester, NY, USA: Cambridge University Press.

GLOSSARY

API	Application Program Interface.
ATM	Automated Teller Machine.
Black-box testing	A type of testing when the inner structure is unknown to the person performing the test. The opposite to black-box testing is white-box testing where the structure is known.
DD	Decimal Degrees. A longitude/latitude coordinate with degrees.
DM	Degrees Minutes. A longitude/latitude coordinate with degrees and minutes.
DMS	Degrees Minutes Seconds. A longitude/latitude coordinate with degrees, minutes and seconds.
Exploratory testing	To sit down and try out the software under test in order to trying to figure out its functionalities and what to test.
GPS	Global Position System.
GPGGA	A GPS message consisting of among other things, a GPS coordinate.
GPRMC	A GPS message consisting of among other things, a GPS coordinate.

GSM	Global System for Mobile Communications.
GUI	Graphical User Interface.
Lateral testing	*See exploratory testing.*
NMEA	A format of a GPS coordinate with degrees and minutes.
OS	Operative System.
PDA	Personal Digital Assistant.
RTOS	Real Time Operative System.
WLAN	Wireless Local Area Network.

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

På svenska
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.
Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.
Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.
För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English
The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.
The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.
According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.
For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/
© Johan Andersson & Katrin Andersson