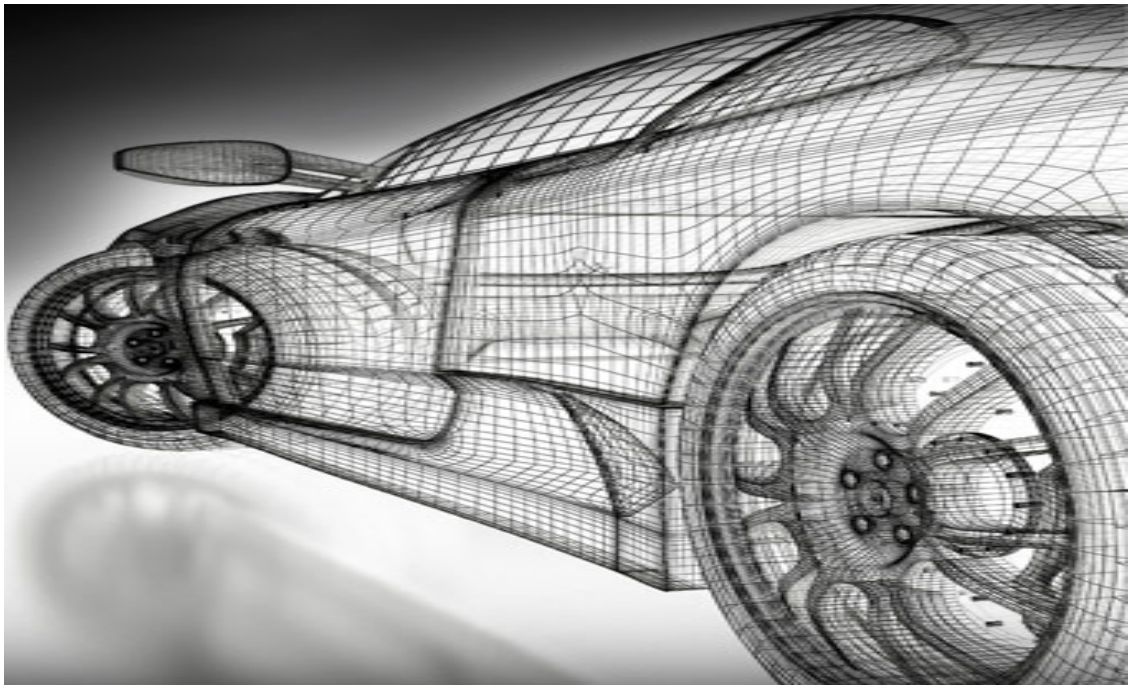
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Controlled Experiment on Coverage Maximization of Automated Model-Based Software Test Cases in the Automotive Industry

Master's Thesis in Software Engineering

RASHID DARWISH
LYNNIE NAKYANZI GWOSUTA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016:NN

A Controlled Experiment on Coverage Maximization of Automated Model-Based Software Test Cases in the Automotive Industry

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

A Controlled Experiment on Coverage Maximization of Automated Model-Based Software Test Cases in the Automotive Industry
RASHID DARWISH
LYNNIE NAKYANZI GWOSUTA

Supervisor: Richard Torkar
Examiner: Miroslaw Staron

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46(0)31- 772 1000

Cover: Computer software is being used to test car performance before prototypes are built.

Typeset in LaTeX
Gothenburg, Sweden 2016

A Controlled Experiment on Coverage Maximization of Automated Model-Based Software Test Cases in the Automotive Industry

RASHID DARWISH
LYNNIE NAKYANZI GWOSUTA
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Background: In the automotive industry, as the complexity of ECU's increase, there is need for creation of models that facilitate early tests to ensure functionality; but there is little guidance on how to write these tests in order to achieve maximum coverage.

Objective: We evaluated our prototype CANoe$^+$ which uses the CANoe and GraphWalker tools Vs CANoe with regard to coverage maximization of generated test cases from the viewpoint of both software developers and software testers. The possibilities and limitations of this approach are also stated.

Method: We conducted a controlled experiment using a nested design with the authors executing sample functions using the prototype and CANoe tools multiple times (240 runs) for each tool. The coverage data from the experimental runs of the two treatments i.e. CANoe$^+$ and CANoe was collected and statistically analyzed.

Results: CANoe$^+$ was significantly more effective than CANoe at an alpha level of 0.05 for a one-tailed test while using the Mann-Whitney-Wilcoxon statistical test.

Limitations: Using the presented approach could be unfeasible if one attempts to test the whole system in one go. It is best suited for when a specific module of the system needs to be tested after which one can move to the next module and then cover the whole system in the long run.

Conclusion: The results reinforced the existing evidence regarding the superiority of using model-based testing techniques like CANoe$^+$ over using testing methods like CANoe in automotive systems.

Keywords: Model-Based Testing, Graph Theory, GraphWalker, CANoe, Transition Based Modelling, Software Testing, ECU, Automotive Industry, Controlled Experiment.

Acknowledgements

I would like to thank our supervisor, Richard Torkar, for his support and guidance throughout this thesis. Special thanks go to our industrial supervisors; Alixander Ansari and Daniel Nilsson for providing valuable input throughout this thesis project. My thanks go to my co-author Lynnie Nakyanzi for her co-operation during this thesis work. Additionally i thank Emma Fornander for the CANoe training she gave to us. Last but not least, I want to thank my parents, my wife Amna, my sister Natalie, my brother Khaled and the family at large for all their love and support.

Rashid Darwish, Gothenburg, June 2016

I would like to express my deepest gratitude to our supervisor, Richard Torkar, who has guided us through the work of the thesis. We thank Richard, not only for his useful remarks on the report, but also for sharing his knowledge within the field and unabridged willingness to aide us during the work on the thesis. I would also like to appreciate our industrial supervisors; Alixander Ansari and Daniel Nilsson for all their support and guidance throughout the project as well as Emma Fornander for the CANoe training she gave to us. My thanks go to my co-author Rashid Darwish for his co-operation during this thesis work.

Special thanks go to the Swedish Institute for granting me a scholarship to study my master program, it will always be much appreciated.

Finally i want to thank my mother, my husband Ivan, my brother Lyndon and my friends for all their love and endless support during my education, without them i wouldn't have come this far.

Lynnie Nakyanzi Gwosuta, Gothenburg, June 2016

Contents

List of Figures

List of Tables

1
Introduction

This chapter presents the Problem Statement, Research Objectives, Context and research methods for this thesis.

1.1 Problem Statement

Today as software in Electronic Control Units (ECUs) gets more and more complex, there is an ever increasing need for efficient testing processes in the automotive industry. ECUs is a generic term for any embedded system that controls one or more of the electrical system or subsystems in a motor vehicle [35]. Automated model based software testing has been proposed but due to the complexity of the systems being built, there is a challenge to generate automated test cases that are effective and have a high coverage [9].

With this in mind, creating a method that enables the developers to model the behavior of the desired system using graph theory techniques and generate automated test cases to intelligently test the system functionality is most sought after. Addressing this challenge will give an insight into the possibilities and limitations of using model based testing with graph theory techniques to generate effective test cases that have a maximum coverage.

The knowledge will be applied in the automotive industry by capitalizing on the advantages of two tools i.e. CANoe[1] and GraphWalker[2] to generate and execute test cases with maximum coverage. By maximum coverage we mean how much of the system's functionality is exercised by the generated test cases.

This will verify the system requirements by the use of models and validate that the system under test meets the customer's needs. Additionally, our assumption is that in the long run this approach will reduce the costs of regression testing[3] and the developer efforts will be channeled to exploratory and negative testing.

1.2 Research Objectives

We aimed to evaluate our built integration prototype of the CANoe and GraphWalker tools when used together to generate and execute tests as well as ensure maximized coverage for the executed tests. We shall subsquently refer to our de-

[1] http://vector.com/vi_canoe_en.html
[2] http://graphwalker.github.io/
[3] https://en.wikipedia.org/wiki/Regression_testing

veloped prototype that uses CANoe and GraphWalker tools as CANoe$^+$. This was carried out from the point of view of both the software developers and software testers in the automotive industry.

With this thesis, we expect to have both an industrial and an academic impact. In terms of academia, we shall address challenges which have previously been identified by empirical studies [4] that relate to maximizing coverage in automated model based testing. Additionally by addressing this challenge, we shall get insights into the possibilities and limitations in regards to this approach as well as aim at publishing this thesis.

As the targeted challenge is highly relevant to industry, successfully addressing it will have an impact on the industry. Furthermore, if the thesis shows enough potential, there is a realistic chance that the proposed solution will be directly transferred to the automotive industry.

1.3 Context

The prototype that was evaluated made use of sample ECU functionality e.g. closing and opening of a car door, starting and stopping the ignition, rolling up and down the windows and turning on and off of lights in a car system that was simulated in the CANoe software.

Andrea et al. notes that empirical studies of randomized algorithms do not involve human subjects and the number of runs (i.e. n) is only limited to computational resources [2]. Further more, they state that the probability distribution of a randomized algorithm can be analyzed by running such an algorithm several times in an independent way, and then collecting the appropriate data about it's results and performance.

Since CANoe$^+$ contains a notion of randomness, we chose to evaluate the prototype ourselves as opposed to using human subjects. The authors acted as the participants and took part in the experiment as they executed the sample functions multiple times while using both the CANoe$^+$ and CANoe tools. Coverage data of the two tools was collected and later used in the analysis and discussion chapters to ascertain which of the two tools gave more advantages and/or disadvantages than the other.

Randomization was used to assign the participants to the treatments in order to assume independence and validity of results. The participants who had an average experience of five years in the software engineering field were involved in the two treatments i.e. the use of CANoe and then the use of the new prototype which is CANoe$^+$.

The prototype (CANoe$^+$) took six person months effort and was evaluated against the current state of practice which is the use of the CANoe tool without the notion of coverage. The participants used the sample functionality for multiple runs of the two treatments and the test metrics from the execution of the functions were noted down.

1.4 Research Methods

In software engineering, empirical methods are crucial, since they allow for incorporating human behaviour into the research approach taken. The main motivation is that it is needed from an engineering perspective to allow for informed and well-grounded decisions to help evaluate as well as validate the research results [36]. Here we describe four research methods; case studies, surveys, post-mortem analyses and controlled experiments. We also motivate why controlled experiments was our best option.

Case studies are conducted to investigate a single entity or phenomenon within a specific time space. They normally study real projects and hence are used for monitoring projects, activities or assignments. Data is collected for a specific purpose throughout the study and it has a low level of control as it is an observational study. An advantage of case studies is that they are easier to plan but the disadvantages are that the results are difficult to generalize and harder to interpret, i.e. it is possible to show the effects in a typical situation, but it cannot be generalized to every situation. Furthermore, researchers are not completely in control of a case study situation. This is good, from one perspective, because unpredictable changes frequently tell them much about the problems being studied. The problem is that one cannot be sure about the effects due to confounding factors. The difference between case studies and experiments is that experiments sample over the variables that are being manipulated, while case studies sample from the variables representing the typical situation. Also one does not have the same level of control over a case study as in an experiment.

Surveys are referred to as research-in-the-large (and past), since it is possible to send questionnaires to or interview large numbers of people covering whatever target population is available. Thus, surveys are often an investigation performed in retrospect, when e.g. a tool or technique, has been in use for a while. The primary means of gathering qualitative or quantitative data are interviews or questionnaires. The advantage is that they are relatively easy to administer since they can be administered remotely and there is also a capability of collecting data from a large number of respondents. The disadvantage is the cost and time, which depend on the size of the sample, and are also related to the intentions of the investigation.

Post-mortem analyses are conducted on the past as indicated by the name but they also focus on a typical situation that has occurred. Thus, post-mortem analyses are similar to case studies in terms of scope and to surveys in that they look at the past. The basic idea behind post-mortem analyses is to capture the knowledge and experience from a specific case or activity after it has been finished. Post-mortem analyses help to collect data and feedback which can be used in the future for improvements. It is easy for the people being analyzed to conceal information out of anxiety about negative consquences, this can greatly affect the analysis being carried out.

Finally controlled experiments are often conducted to compare a number of different techniques, methods, working procedures and are appropriate when testing the effects of a treatment. In an experiment, the state variable can assume different values and the objective is normally to distinguish between two situations. The

researcher has control over the study and how the participants carry out the tasks that they are assigned to. More so, the study can be planned and designed to ensure high validity, although the drawback is that the scope of the study often gets smaller.

We chose to carry out a controlled experiment as we wanted to have control over the participants and since the prototype involved random algorithms, executing it several times made more sense for an experiment to be carried out while ensuring high validity. Most importantly, the results of an experiment are usually more generalizable and experiments have a high level of replication than any of the other research methods.

The rest of the report is structured as follows; we first describe the background (Chapter 2) ; a description of our developed prototype follows in Chapter 3. Chapter 4 describes the most relevant related work to our study and next the design of the experiment which lists the hypotheses in (Chapter 5). Chapter 6 then follows with the analysis of the collected data and treatment of the results. The Discussion is presented in Chapter 7 stating the evaluation of results and Chapter 8 states the threats to validity. Conclusions are drawn in Chapter 9.

2
Background

Model Based Testing is the process of test generation from models of/related to a system under test (SUT) by applying a number of sophisticated methods. The basic idea of model based testing is that instead of creating test cases manually, a selected algorithm automatically generates test cases from a model [4]. This reduces the test design time and allows for the generation of a variety of test suites from the same model simply by using different test selection criteria among others.

With the use of a model-based testing tool, test cases are generated from the abstract model of the software under test and the test cases are implemented into executable tests that are later executed automatically using the selected algorithm. Test reports are also generated from the resulting comparison between each of the test outputs from the software under test with each of the expected outputs [32].

The model-based testing process consists of several main steps i.e. modelling of the software under test, generation of abstract tests from the model, concretization of the abstract tests to make them executable, execution of the tests against the software under test (SUT) to assign verdicts and the analysis of the test results.

The first step, modelling of the software under test, is to write an abstract model of the system that is to be tested based on an actual model as in figure 2.1. The abstract model has to focus on just the functionality the test developers want to test and abstract away other details that are not to be tested. After describing the model, it can be checked for verification and validation of which GraphWalker can be used to this effect.

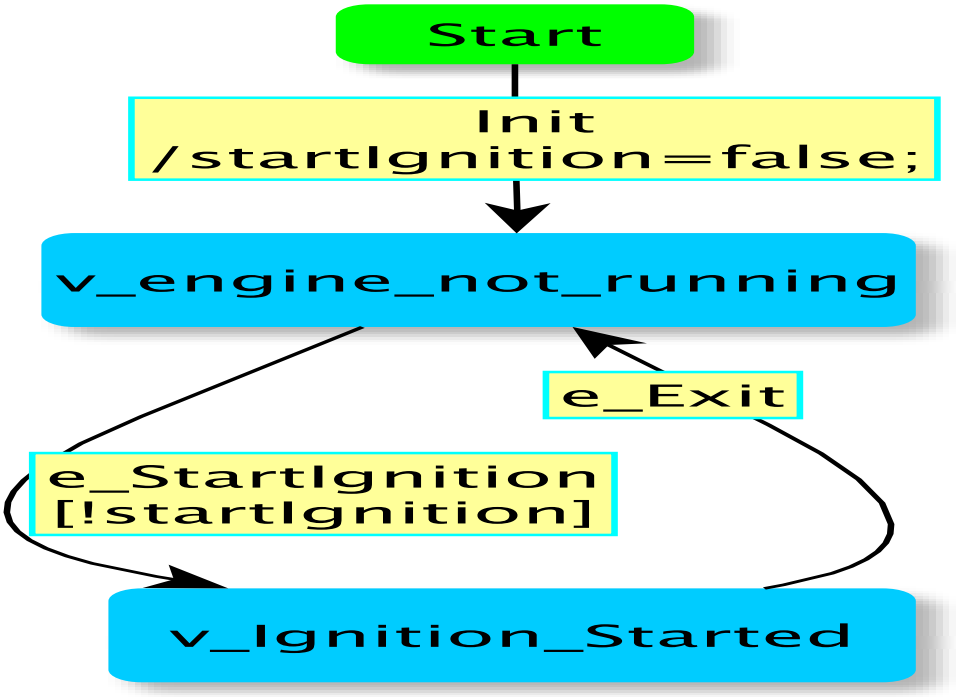


Figure 2.1: Sample model for ignition functionality

The second step involves generation of abstract tests from the model. The abstract tests are generated automatically and are a simple view of the software under test. Hence, they don't contain detailed information on how to execute test cases directly but are an interface of the methods to be implemented before the tests can be executed. The test developer then has to decide on the test selection criteria which also determines the test coverage to identify which tests one wants to generate from the model as there can be an infinite number of possible tests. The output from this step are the abstract tests which are sequences of operations from the model.

Listing 2.1: Sample interface that is generated from the model in Figure 2.1

```
// Generated by GraphWalker (http://www.graphwalker.org)
package org.myorg.testautomation;

import org.graphwalker.java.annotation.Model;
import org.graphwalker.java.annotation.Vertex;
import org.graphwalker.java.annotation.Edge;

@Model(file = "org/myorg/testautomation/ignition.graphml")
public interface ignition {

@Vertex()
void v_Ignition_Started();

@Edge()
void e_Exit();

@Edge()
void e_StartIgnition();

@Vertex()
void v_engine_not_running();

@Edge()
void Init();
}
```

The third step is to transform the abstract tests into executable concrete tests. This can be done by the use of a transformation tool which translates each abstract test into an executable test script. This bridges the gap between the abstract tests and the concrete software under test by adding in the low-level SUT details that were not mentioned in the abstract model.

The fourth step is to execute the concrete tests against the system under test. This can either be done online where a model-based testing tool connects directly to a SUT and tests it dynamically or offline where a model-based testing tool generates test cases as computer-readable assets that can later be run automatically.

Lastly, the analysis of results from the test execution and taking of corrective action. For each failed test, the fault that caused the failure must be determined. When a test fails, it may be due to a fault in the SUT or it could be a fault in the test case itself. Nonetheless, we are able to get feedback on the correctness of the model as faults can either be tied back to the model or to the executable tests. Some of the advantages of model based testing include; Fault detection in the SUT, reduced cost and time for testing, improved test quality, detection of requirements defects, traceability and requirements evolution [4].

2.1 Technology Under Investigation

The main investigation involved generating tests from models, implementing the tests and then executing the tests for maximized coverage. Different tools were used to achieve this and they include;

2.1.1 Transition Based Modelling with (yEd Graph Editor)

yEd is a desktop application that aids users to quickly and effectively create high quality diagrams which we call models; in our case finite state machines or extended finite state machines. It models the software under test as states and transitions which means that the system under test can be in a finite number of different states and the transitions from one state to another are determined by the rules of the machine. It saves the models in GraphML format which are later used in the GraphWalker tool [40].

2.1.2 GraphWalker

GraphWalker is a model based testing tool built in java that uses the command line. It reads models in the form of finite state diagrams or directed graphs and generates tests from the models, either offline or online [11]. These models are saved in the GraphML format and are created using the yEd tool. GraphWalker also provides a way to check the model to ensure there is at least one direct path from the start state to the end state. Additionally, it provides different algorithms to generate test cases and can also generate skeleton code for test adapters using a default or user provided code template. An interface is available to the user to describe the adapter behavior, including the software under test. In our case, GraphWalker uses CANoe as the test driver to execute the test cases i.e. GraphWalker aides CANoe in the generation of the different test sequences that are executed by CANoe. The results of the execution are shown in a report that is generated by CANoe.

There are a number of reasons for using GraphWalker as the model based testing tool of choice. These include;

- Modeling using a finite-state diagram is visual, it's easy to understand. Getting feedback from team members and stakeholders is so much easier with models.
- The models create an abstraction layer between the test design and the implementation of automation code. This is important when it comes to maintenance.
- GraphWalker is very easy to setup and start using in your test automation code. More so, it's highly modularized, and easy to extend if you need to customize it.

2.1.3 CANoe

CANoe is the most widely used comprehensive software tool for development, test and analysis of entire ECU networks and individual ECUs in the automotive industry today. It provides support throughout the entire development process - from

planning up to final system-level tests [33]. In our case, CANoe was used as a test driver to execute the tests of the system under test which is the simulated network or signals of the ECU.

CANoe stores the actual values of our tests which we compare against the inputs in the model. It runs the test sequence as provided by GraphWalker to generate the test report. Responses from the SUT and from the model can then be compared by means of test cases as executed by the sequencer(test generation algorithm i.e. functional test). If the response from the SUT is the same as the output from the model (according to a defined signal comparison method), the test case is passed or failed otherwise.

An aspect of testing with models worth mentioning is that very often this type of testing enforces what is referred to as gray-box testing; i.e. internal model signals are compared with internal SUT signals [4].

2.2 Alternative Technologies

In the recent past, the development of automotive embedded devices has changed from an electrical/mechanical engineering discipline to a combination of software and electrical/mechanical engineering. The effects of this change on development processes, methods and tools as well as on required engineering skills has been very significant and is still ongoing today. At the present, the trend is to use model-based development in the automotive industry as models improve reliability of systems and facilitate reliability measurements [27].

Software components are no longer handwritten in C or assembler code but modeled with MATLAB/Simulink trade, Statemate, or similar tools. However, quality assurance of model-based developments especially testing is still poorly supported [4]. Traditional tests that are still in use suffer from,"pesticide paradox" because the tests become less effective at catching bugs as the testing process progresses [27, 26] . The software under test is constantly changing in functionality while the traditional tests are static and they need to be adapted to the new behavior. The maintenance of the static tests could be costly [27, 26] .

With this realization, model based testing has become a niche for testing systems as it has been used in industries like telecommunications and avionics which have a stringent software quality bar with reported success [4, 27]. Model based testing tackles the above challenges by explicitly describing the software under test behavior to generate and maintain useful, flexible tests [27, 26]. One distinct advantage of model based testing shows up when measuring coverage; it is not possible to execute all combinations of test paths. However, it is feasible to measure what part of the model has been covered when generating test cases with graph traversal algorithms [27].

Testing tools for example TPT (Time partition test) tools have been developed based on graphical test models but such tools are limited to supporting major test activities, aiding the selection of test cases, formation of simple representations of test cases and providing an infrastructure for automated test execution. However, there is still a long way to a fully integrated and feature rich testing technique for model based testing in the automotive domain that utilizes models as the basis for

integration tests let alone aide in regression testing [4].

In another study where event sequence graphs were used, test cases representing complete sequences of events were automatically generated basing on event sequence graphs. Also the test cases were minimized by using the chinese postman problem [26] and later transformed into test scripts for CANoe which were then applied to exercise the SUT. This approach significantly reduced the time to implement the test cases and their execution. By using the approach, generation of test cases was fully automated hence simplifying regression testing and a new fault which had not been previously detected with the classification tree method was uncovered. The only downside to this approach was the amount of time it took to create the event sequence graph models. Lastly, this approach requires the generated test cases from the event sequence graph models to be automatically translatable to a format that is readable by the CANoe tool that is used within the test environment [3].

The real challenge here was to address the maximization of coverage by interfacing two tools ie GraphWalker based on graph theory techniques and CANoe that is used for test execution in the automotive industry which to the best of our knowledge has not been done before. In our opinion, we think that it would be the best of both tools if there is an interface that integrates the two tools. Effective test cases would be generated from models that would then be executed automatically for maximum coverage. The integration has a number of advantages which include;

- Different paths through the model are executed hence making the testing process more effective.
- Different personnel can work on different levels of abstraction i.e. the modellers, the software programmers and the software testers.
- It saves time during regression testing as with CANoe$^+$ changes to values or improvements can be made in the model and the variables in the code base are automatically updated and executed.
- Lastly, it helps in visualization between the different stakeholders as the model can be widely understood by all. Chaudron et al. states one benefit of modelling as the stakeholders have a shared common representation of the system being built among others [5].

2.3 Relevance to Practice

GraphWalker as a tool has been applied to a number of software projects where it helps engineers generate offline or online test cases from finite state machines and extended finite state machines. Online test cases are automatically generated and executed on the fly while offline tests are generated as computer readable assets that can later be run automatically. It reduces the efforts that are needed by engineers in testing through the automation of tests which can also be run as regression tests. It has also been used in tree traversals where it keeps track of the visited nodes [19].

GraphWalker has been used in safety automata like in [10] for test case generation where different coverage criteria were used to that effect. They went ahead to focus test case generation on specific parts of the safety automaton where the test case length was limited to 50 test steps so as to have full transition coverage. Since the generated test cases covered all transitions and hence all states, it became

much easier to detect missing states or transitions as well as erroneous transition conditions.

Additionally, GraphWalker has been integrated with keyword and behavior driven framework and Robot framework. The integration had positive results as there were improvements in the flexibility, maintenance and coverage as a result of using GraphWalker [29].

Lastly, it has also been used in mobile systems [12] to generate abstract tests and also to traverse the models for 100% coverage of states and transitions. Many faults were discovered in an application that had already been tested and this goes on to assert that GraphWalker produces the necessary coverage to find most if not all faults and bugs in systems. 100% coverage does not mean the system is free from bugs or faults it just implies that all the transitions through the model have been covered.

A survey [1] lists CANoe among the tools used in the automation industry during test design and the test execution phase which offers test environments and simulation capabilities. It further states that it offers graphical editors to design the tests and setup the execution after which it generates a test report. CANoe helps engineers in simulation of ECU functionality as well as in running tests which later generate test reports with the verdict of the software or system under test.

Like GraphWalker, CANoe has been used as an automated test execution tool for test cases in several projects; [3, 23, 21, 38] where ECU signals are simulated and diagnostics validated. After execution of the tests, the tool generates a test report indicating the test cases that have passed and or failed.

Other tools for example Petra [30] which is used by Porsche have been developed based on CANoe to validate gateway ECU's. The tool automatically generates gateway tests from routing tables and the tests are later used in driving trials or to check the routing functions online.

3

CANoe+

CANoe+ is the proposed solution to solve the challenge at hand and it is the result of intergrating the GraphWalker and CANoe tools. An interface was implemented with the aim to act as a communication bridge between the two tools. However, in addition to the above tools, the solution includes a model and test Modules. There are three major steps involved when writing tests with CANoe+ as can be seen in the context diagram below;

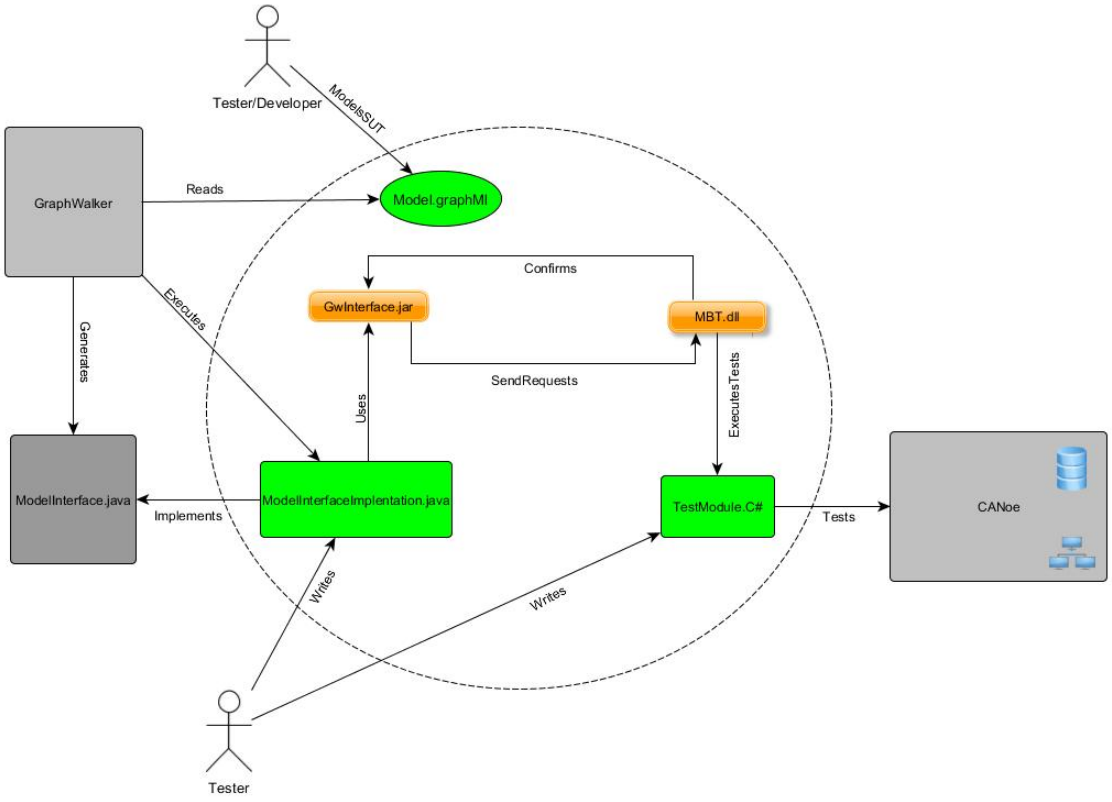


Figure 3.1: Context diagram for CANoe+

3.1 Modelling

CANoe+ involves modelling of the functionality that is to be tested. Using the yEd desktop application, a model of the expected behavior of the system under test is

drawn as an extended finite state machine. It models the system under test as states and transitions, the model is saved in the GraphML format. The saved model is provided to the GraphWalker tool as an input of the expected states, transitions and the values that need to be tested. The model that we drew can be seen in Appendix A.

3.2 GraphWalker

In our context, GraphWalker has three major roles;

- Checks the model to ensure that there is atleast one direct path from the start state to the end state.
- Generates an interface i.e. abstract tests from the provided model.
- Executes the tests in an non-deterministic sequence.

GraphWalker is connected to CANoe by the communication bridge mentioned above and for the purpose of this communication, the interface that is generated by GraphWalker is implemented and it uses the "gwInterface.jar". We used the 100% edge coverage selection criteria to enable all the edges to be covered which in turn leads to the coverage of the states. The jar file has one single task when GraphWalker is executed, which is to deliver a GraphWalker request to a server which in its turn executes the test. The GraphWalker request includes a snapshot of the current state/transition and it's values. The snapshot is packed in a "JSON" object.

3.3 CANoe

CANoe includes a simulated CAN network, and a database which holds the values of the network. The simulated CAN network in our context is the system under test. CANoe is responsible for the test framework and it is the test driver that we use to test the system under test. We used a .Net test module and it's libraries provided by CANoe to implement the test module. The test module is the implementation of the adapter to the SUT, which should correspond 100% with the interface generated by GraphWalker in the previous step. The test module uses "MBT.dll", which is the other end of the bridge mentioned above, and acts as a server to execute the requests received from GraphWalker.

3.4 Execution

GraphWalker uses random functions to generate random test sequences through the model that ensure edge coverage and hence when it is executed, it sends a request for a given function to be executed. While sending the request, it has the expected value appended to it. When the function to be executed in the test module is found, a comparison is made between the expected value from the model and the actual value from the CANoe database. A confirmation is sent back to GraphWalker to affirm that the functionality was executed and also CANoe then produces a test

report showing the verdict of if the function which is referred to as the testcase passed or failed.

3.5 Effort and cost considerations

To use CANoe+, one needs to model the the system under test, add the gwInterface.jar to GraphWalker and use the interface it provides to implement the executable tests . At the CANoe side , the MBT.dll library is added into the test module and the generated interface is implemented as a .NET test module.

Incase modifications or changes need to be made in the system under test, for example if a variable is defined in the model and one needs to use it in the test, all that needs to be done is to define it in the java class that implements the generated interface and in the .NET test module that implements the generated interface as well. That is all that is needed.

With this approach, regression testing becomes easier as changes and functionality are easy to incorporate and write tests easily at a much lower cost.

4

Related Work

There has been substantial research in the area of automated test case generation and model based testing in general but below, we briefly describe the most relevant studies to our thesis and their findings.

4.1 Related Work

A study by [22] shows that in model based testing, models form the basis for generating test cases which can be used to show model-Code conformance. It also notes that coverage achieved by individual test generation techniques in a broader sense complement each other hence the need to integrate the different test generation techniques. An integrated test generation tool SmartTestGen that uses various test generation engines with each engine implementing a different technique was developed. Simulink stateflow models were used for modeling and it was evaluated against REACTIS and embedded tester. The tool takes models and a test specification as input and produces a test suite as output. From the evaluation, it can be noted that coverage increases when tests are automatically generated and coverage is evaluated in three ways ie decision coverage, condition coverage and modified condition/decision coverage.

[20] looks at an approach to automatically generate a small number of test cases that cover all reachable states in closed loop controller software without losing the coverage of any state. In their approach they assume one has access to source code hence they focus on path coverage since covering all reachable paths through the code would imply covering all reachable states. The goal is not to achieve 100% path coverage through the whole program but rather to cover 100% of the sub paths through the various functions. Their results demonstrated that their approach could reduce the number of test cases by tens of thousands and test generation time by dozens of hours with no negative impact on the fault - finding capability.

Also [14] describes how mutation testing suffers from high computational cost of automated test-vector generation due to a large number of mutants that can be derived from programs and the cost of generating test-cases in a white box manner. A novel algorithm is proposed for mutation -based test case generation for simulink models that combines white box testing with formal concept analysis. By exploitation of similarity measures on mutants, it is possible to generate small sets of short test cases that achieve high coverage on a collection of simulink models from the automotive domain. Even though the paper focusses on dataflow models given in the simulink design language, test case generation for simulink models is complicated

by the fact that the simulink language lacks a formal semantics and makes heavy use of floating -point arithmetic. Their results indicate that using their algorithm, the smallest number of test cases is produced along with the best coverage.

Some studies have pointed to the use of statistical tools and methods like markov chains [13, 7] to generate and execute tests from software specifications. MaTeLo uses markov chains as the usage model for software applications to generate test cases. The key point of the approach is no longer oriented to ensure that all the code has been successfully tested, but more focused to know and model the future usage of the software application in order to guarantee the use of the released software without failure. In some cases [28] even before the test cases were executed, deficiencies and ambiguities in the system specifications were identified. The MaTeLo tool further provides metrics that could help technical staff determine the software quality and evaluate how much of the expected results are met.

MTest [17] combines module tests with model-based development. The central element is the classification tree method which is mostly used for C-code testing and has now been adopted to the needs of a model-based development process for embedded systems. MTest is used to test automotive software from model-in-the-loop over software-in-the-loop down to processor-in-the-loop testing. Additionally, test scenarios once developed can be reused in a hardware-in-the-loop environment thus, providing a means to automatically test automotive software within the whole development process.

A control function to be developed is described by the means of simulation tools like MATLAB/Simulink/Stateflow (function design). Based on the interface of the logical model, and by using the classification-tree method, the function developer can derive test scenarios systematically and describe them graphically. With the graphical representation the user gets visual information about the test coverage which indicates, how well the test cases cover the range of possible test input combinations. Once the tests have been executed, a report is instantly generated and displayed with the results structured hierarchically and displayed as a tree. The result tree can include any data item and any test which has been done in the different simulation modes. The user can navigate through the tree and view all details.

The MB^{3}T [6] approach was developed to minimize the challenges of the test development process by creating a systematic procedure for the design of test scenarios of embedded automotive software and its integration in the model-based development process. Test scenarios are defined in two different perspectives i.e. requirement-based test design and model-based test design with a consistency check to create consistency between them. According to the approach, logical test scenarios are first defined based on the textual requirements specification of the embedded software. The test scenarios are specified at a high level of abstraction and don't contain any implementation details of the test object. Due to their close link to the requirements, it is easy to check which requirements are covered by which test scenario. Subsequently, the requirement-based logical tests are refined to executable model-based test scenarios. Also the approach helps to check, whether or not the logical test scenarios are fully covered by the executable test scenarios. The requirements based design helps to check necessary requirements coverage while the model-based test design with the classification tree method for embedded systems

guarantees the systematic derivation of time-variant scenarios for the executable artifacts of model-based development. The comparatively higher effort, resulting from the tests being designed from two different angles, leads to a better and more systematic way of generating tests which, in turn, ensure a test coverage with regard to two complementary coverage criteria.

A case study [25] where an automotive network controller was used to assess different test suites in terms of error detection, model coverage and implementation coverage. An experiment was setup where requirements documents were used to build an executable behavior model of the network controller whereby this process revealed inconsistencies and omissions in the specification documents that were updated accordingly. The models were then used by the developers, test engineers and different engineers who both manually and automatically derived tests on the grounds of the model. Some of the test suites were generated automatically with and without models, purely at random, and with dedicated functional test selection criteria. Other suites were derived manually, with and without the model at hand. Both automatically and manually derived model-based test suites detected significantly more requirements errors than handcrafted test suites that were directly derived from the requirements.

It was found that tests derived without using a model detected fewer failures than model-based tests. The number of detected programming errors was approximately equal, but the number of detected requirements errors—those that necessitated changing the requirements documents—was higher. Hence automatically generated test suites detect as many failures as handcrafted model-based test suites with the same number of tests. A sixfold increase in the number of automatically generated tests leads to 11% additionally detected errors. None of the test suites detected all errors.

vTESTstudio [39] is a high performance development environment for creating test sequences which can be used in all product development phases. It generates test sequences which can be executed with the CANoe test sequencer in real-time and evaluated in detailed reports. It seamlessly intergrates both proven and new types of test design methods and test notations. Graphical test diagrams i.e. flowcharts are used to model sequences of tests and even though its close to our prototype, it does not generate random test sequences which are crucial in uncovering hidden bugs.

5

Experimental Design

We chose to carry out a controlled experiment to evaluate the prototype because we had control over which participants were to use the prototype, when and where it would be used so as to facilitate in the generalization of results [16]. The prototype evaluation was carried out by the authors who executed the software multiple times while using the CANoe$^+$ and CANoe tools. We followed [16] guidelines on reporting experiments in software engineering which aims to make reporting styles homogeneous and to aide in the integration of experiment results into a common body of knowledge as well as to support readers.

5.1 Goals

With the above in mind, the goal of the experiment was to evaluate our developed prototype (CANoe$^+$) and assess if it increased the coverage of automated model-based software test cases in the automotive domain while studying its possibilities and limitations.

Since no solution that uses the CANoe and GraphWalker tools has been developed yet, we intended to evaluate our prototype against the current way of testing software which is in use. CANoe$^+$ and CANoe are going to be referred to as the treatments subsequently. Our aim was to answer the question;

Does the use of CANoe$^+$ in automotive systems increase coverage of test cases as compared to the current way of testing? The current way of working involves writing tests which are executed sequentially and automatically by CANoe.

5.2 Experimental Units

The authors who are both master students in software engineering and had been using CANoe for close to five months were the participants. Their mean years of working in the software engineering field is five years. Both the authors were assigned randomly by tossing a coin to the treatments i.e. they both used CANoe and CANoe$^+$ to give results. The authors chose to participate in the experiment because CANoe$^+$ comprised of random algorithms which were used to generate random test sequences that were executed by CANoe. The sample functions had to be executed several times to increase validity of the experiment. Futhermore, we aimed to show with high confidence that the obtained results were statistically significant and to assess the performance for each of the tools.

5.3 Experimental Material

The participants used the four sample functions to exercise the CANoe$^+$ and CANoe tools. They were running CANoe 8, Visual Studio 2012 and Eclipse in order to accomplish this task. The functionality to be tested included; closing and opening of a car door, starting and stopping the ignition, rolling up and down the windows and turning on and off of lights in a car system that was simulated in the CANoe software.

During the multiple runs, the test metrics for each test suite i.e. the total number of test cases executed, passed and failed were noted and later analyzed. The sample software was executed in a total of 480 test runs; 240 runs for each tool, 3 faults injected and the coverage metrics noted. This was done in order to ascertain which treatment was better at catching faults. The effort required to use each of the tools was also taken into consideration as we used [18] as a guide to assess the workload of the participants so as to reach a conclusion on which tool required the most effort. It was based on the NASA Task Load Index [34] as attached in Appendix B .

5.4 Tasks

After the test environment was up and running the software was loaded into the testing environment and tests executed to ascertain the effectiveness of fault discovery and coverage by the two treatments. The participants also injected faults and the coverage measurements were noted for each fault and if there were any faults at all.

5.5 Hypothesis, Parameters and Variables

The experience of the participants was used as the blocking factor to increase the validity of the experiment since we were only interested in investigating to see whether CANoe$^+$ produces better test coverage than CANoe.

The participants were sampled randomly over the two treatments to prevent variability from biasing the results. The factor we considered for this experiment was;

- Coverage of the tools i.e. we compared the use of CANoe$^+$ to the use of CANoe.

The dependent or response variable which was coverage facilitated us to know how much of the given model was covered by the tests we created. Coverage is measured in three different aspects ie the smoke test, functional test and stability test. But we focussed on functional tests which depend on 100 % coverage of the edges in the model to fully exercise the software under test and hence determine how much of the software had been exercised. Since we had multiple runs, the mean coverage for the runs were calculated.

Below we formalize our hypotheses:

$$H_0 : MCc = MCc^+$$
$$H_1 : MCc < MCc^+$$

Null Hypothesis: There is no maximization in the coverage of automated model based software test cases produced using CANoe$^+$ tools as opposed to using CANoe.

Experimental (alternative) Hypothesis: The coverage of automated model based software test cases is maximized by the use of CANoe$^+$ tool as opposed to using CANoe.

Using the collected data, the hypotheses were able to give us a basis on whether to reject or fail to reject the null hypothesis.

5.6 Design

Since we had control over the application area, system under test type, developers experience with the language, tools that were used; we used the nested design with the participant's experience as the blocked variable to eliminate bias and experimental error. We chose to use the nested design so that we would not have any interaction effects that were not needed [24] as we were only interested in if CANoe$^+$ produced better coverage of tests than CANoe. More so, the participant's were assigned randomly to the treatments within each block. Repeated measurements were also used to validate the results.

	Coverage	
	Tool Usage	
	CANoe	**CANoe⁺**
Run 1	Functions 1,2	Functions 3,4
Run 2	Functions 3,4	Functions 1,2

Figure 5.1: Nested design for coverage and tool usage showing how the functions will be crossed in the first two runs and subsquently

As shown in the above figure coverage for each of the tools was evaluated in a nested design by use of four different sample functions. We had one level for coverage and two levels for the tool usage i.e. CANoe$^+$ and CANoe. As mentioned earlier due to the randomness of CANoe$^+$ the participants executed the software several times hence yielding a completely related-within subjects design. Threats to validity like experimenter expectancies were ruled out since random algorithms were used for execution in the prototype.

5.7 Procedure

The participants were running CANoe and CANoe$^+$ on different workstations where they used the sample software functions for execution. During the execution, test metrics were noted for each test suite i.e. the total number of test cases executed, passed and failed test cases. At the start of the experiment, each of the two experimenters agreed on how the execution was to be done, when and how the faults were to be injected.

5.8 Analysis Procedures

We chose our analysis techniques while considering the nature of the data that had been collected, the reason for performing the experiment and the type of experimental design that we used. After the experiment was concluded, analysis of the data was carried out. We performed rigorous statistical testing and also measured effect sizes together with the confidence intervals [24].

Since we were trying to confirm a theory that test cases generated while using CANoe$^+$ tool have an increased coverage than those run while using CANoe, the analysis approach that we chose was the analysis of variance. We considered two populations ie those that used the CANoe method and those that used CANoe$^+$, a statistical test was performed to see if the difference in treatment results was statistically significant.

6
Analysis

This chapter presents the descriptive statistics, data set preparation and hypothesis testing of the data collected. It also summarizes the data collected and its treatment.

The sample functions were executed in 240 runs for each tool while recording the number of passed, failed and total test cases with respect to 100% edge coverage of the model. The collected data is shown in Appendix C. Within these runs, there were three fault injections of which data was also collected. The subsequent sections are presented with regards to five scenarios i.e. run without faults injected, run with one fault injected, run with two faults injected, run with the custom fault and lastly a general conclusion with all the collected data. Each of the first four scenarios had 120 independent runs for both the CANoe$^+$ and CANoe tools ,the general conclusion section had all the 480 runs that had been executed in order to validate the data.

The focus was mostly on the number of failures that were reported for each individual run as this gave an overview of how effective each of the tools was at catching faults hence translating into the test coverage for the software under test.

6.1 Descriptive Statistics

Using the R programming language, we were able to compute the descriptive statistics for our data. The `describe()` R function was used on the number of failed test cases for each run to give the measures of central tendency and dispersion for each of the run scenarios i.e. without faults, with one fault, two faults, custom fault and the general conclusion.

6.1.1 Without Faults

In this scenario, no faults were reported using the two tools hence the descriptive statistics were not meaningful to be reported.

6.1.2 One Fault

	Method	
	CANoe	**CANoe$^+$**
Vars	1	1
n	60	60
Mean	20.35	27.18
Sd	15.63	16.85
Median	32	24
Trimmed	21.33	25.4
Mad	0	11.12
Min	0	2
Max	37	111
Range	37	109
Skew	-0.53	2.2
Kurtosis	-1.73	8.34
se	2.02	2.18

Table 6.1: Descriptive statistics of CANoe and CANoe$^+$ with one fault injection. CANoe$^+$ has a higher mean of 27.18 as it finds more faults as compared to CANoe whose mean is 20.35.

6.1.3 Two Faults

	Method	
	CANoe	**CANoe$^+$**
Vars	1	1
n	60	60
Mean	14.42	43.42
Sd	16.03	21.89
Median	1	40
Trimmed	13.81	40.94
Mad	1.48	18.53
Min	0	11
Max	37	109
Range	37	98
Skew	0.27	0.99
Kurtosis	-1.95	0.63
se	2.07	2.83

Table 6.2: Descriptive statistics for CANoe and CANoe$^+$ with two fault injections. In comparison to Table 6.1, Mean for CANoe(14.42) decreases as less faults are discovered whereas CANoe$^+$ mean(43.42) increases as more faults are discovered.

6.1.4 Custom Faults

When a custom fault is injected in the functionality, the CANoe tool is not able to catch this fault and it can never find the fault but CANoe$^+$ catches the fault as expected.

6.1.5 General Conclusion

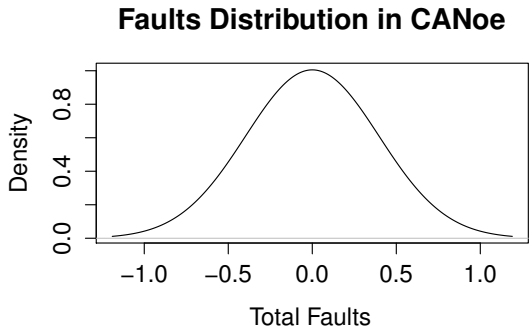
	Method	
	CANoe	**CANoe$^+$**
Vars	1	1
n	240	240
Mean	8.69	19.96
Sd	14.28	21.9
Median	0	14
Trimmed	6.76	16.34
Mad	0	20.76
Min	0	0
Max	37	111
Range	37	111
Skew	1.05	1.5
Kurtosis	-0.89	2.62
se	0.92	1.41

Table 6.3: Descriptive statistics for CANoe and CANoe$^+$ for the general conclusion. All the collected data is used in this scenario to describe the statistics. As seen from the mean, CANoe$^+$ still has a high mean of 1.96 due to the number of faults discovered to the disadvantage of CANoe whose mean is 8.69.

6.2 Dataset Preparation

The collected data was further prepared by drawing box plots and density plots for the number of failed test cases in each run. The R functions `plot()` and `boxplot()` were used to show the distribution of the failed testcases and help notice outliers in the data respectively. This section also analyzes the data in five scenarios i.e. without faults, with one fault, two faults, custom fault and general conclusion. We present a density plot and a box plot side by side where applicable for the scenarios that were run.

6.2.1 Without Faults

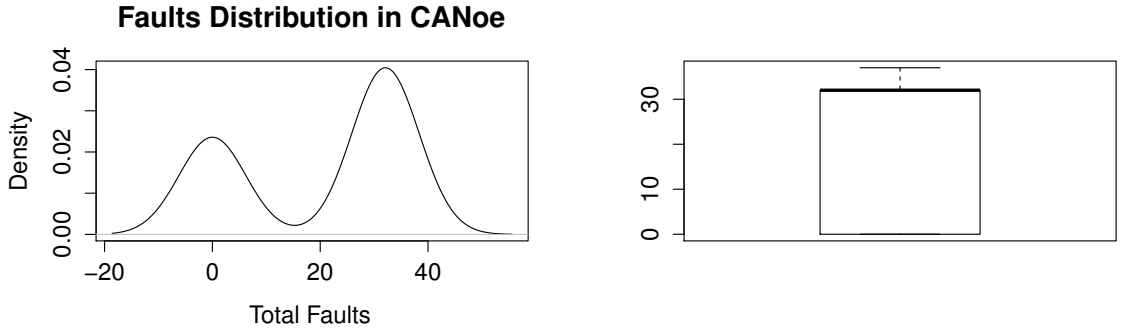


(a) Density plot for CANoe

Figure 6.1: This is the density plot for failed testcases in CANoe and CANoe$^+$ without faults. No faults were discovered using both the tools hence the distribution of the density plot.

6.2.2 One Fault

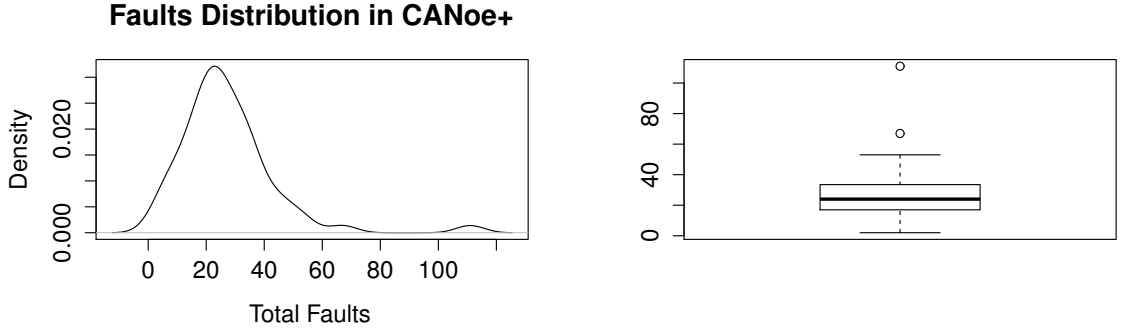
CANoe



(a) Density plot for CANoe **(b)** Box plot for CANoe

Figure 6.2: These are the density and box plots for failed testcases in CANoe with one fault injection.

CANoe$^+$

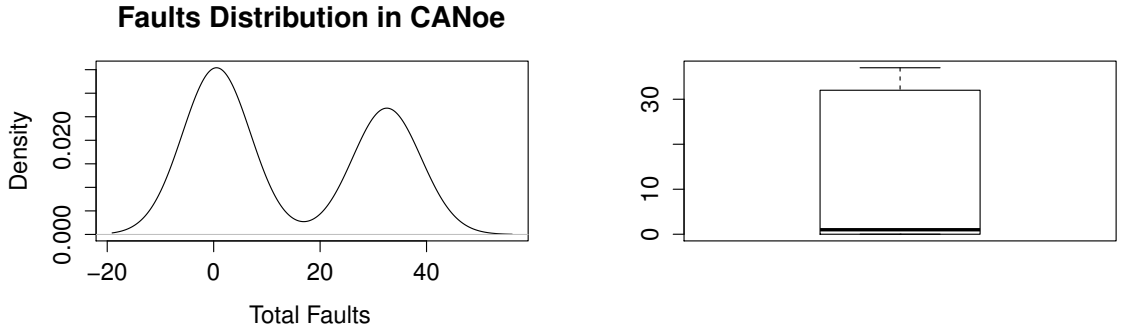


(a) Density plot for CANoe$^+$ **(b)** Box plot for CANoe$^+$

Figure 6.3: These are the density and box plots for failed testcases in CANoe$^+$ with one fault injection.

6.2.3 Two Faults

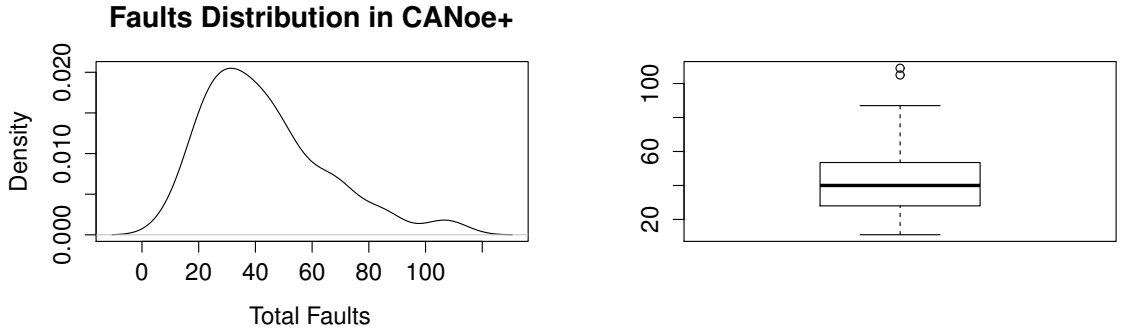
CANoe



(a) Density plot for CANoe **(b)** Box plot for CANoe

Figure 6.4: These are the density and box plots for failed testcases in CANoe with two fault injections.

CANoe$^+$



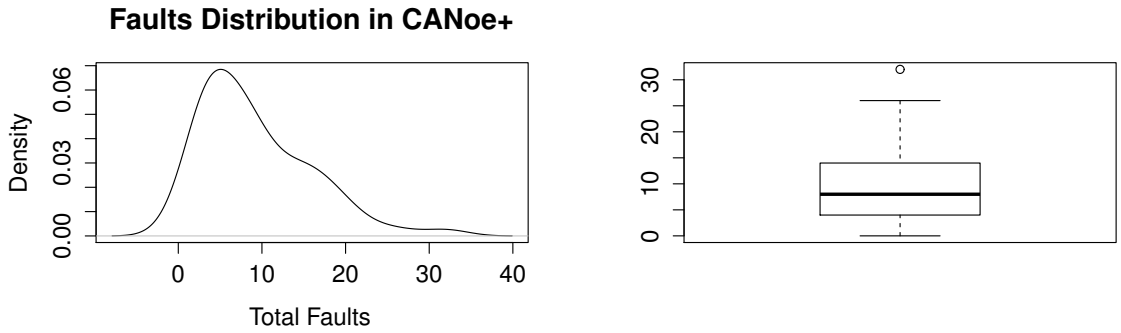
(a) Density plot for CANoe$^+$ **(b)** Box plot for CANoe$^+$

Figure 6.5: These are the density and box plots for failed testcases in CANoe$^+$ with two fault injections.

6.2.4 Custom Fault

In this scenario, the collected data from using the CANoe tool is not described since it does not discover any faults.

CANoe+

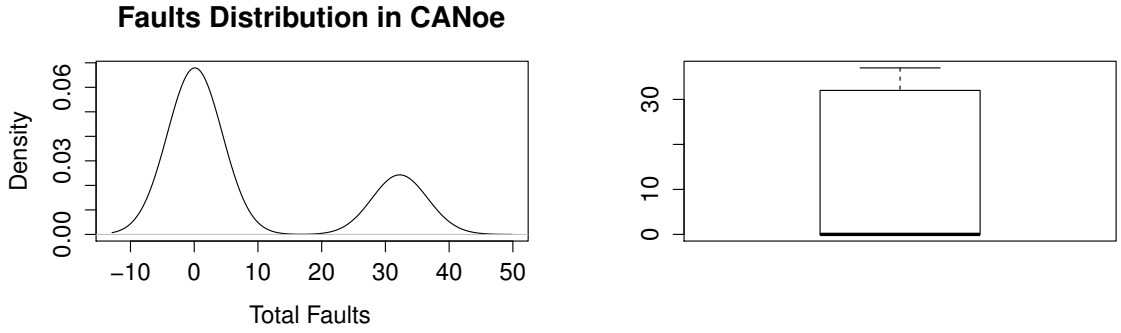


(a) Density plot for CANoe$^+$ **(b)** Box plot for CANoe$^+$

Figure 6.6: These are the density and box plots for failed testcases in CANoe$^+$ with a custom fault injection.

6.2.5 General Conclusion

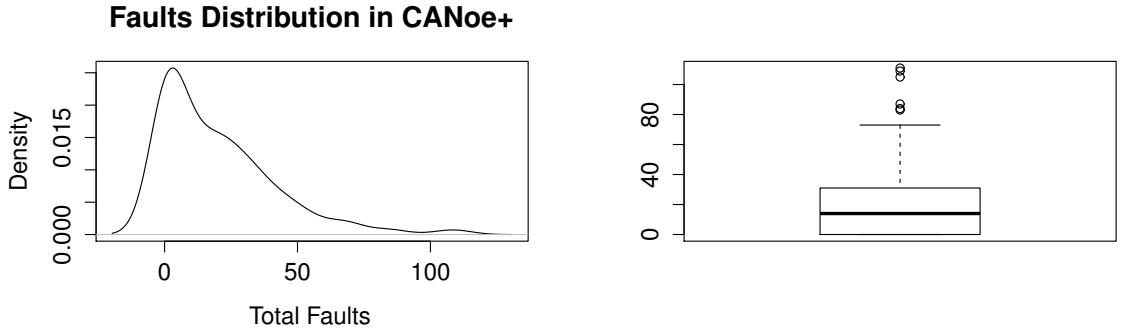
CANoe



(a) Density plot for CANoe **(b)** Box plot for CANoe

Figure 6.7: These are the density and box plots for failed testcases in CANoe for the general conclusion

CANoe$^+$



(a) Density plot for CANoe$^+$ **(b)** Box plot for CANoe$^+$

Figure 6.8: These are the density and box plots for failed testcases in CANoe$^+$ for the general conclusion

6.3 Hypothesis Testing

To analyze the chosen analysis model, we ran the following R functions ; `shapiro.test()` to test for the normality of the sample distribution of the failed test cases, `qqnorm()` and `qqline()` to calculate the quantiles from the normal distribution and to add a line to the Q-Q plot which passes through the quantiles respectively.

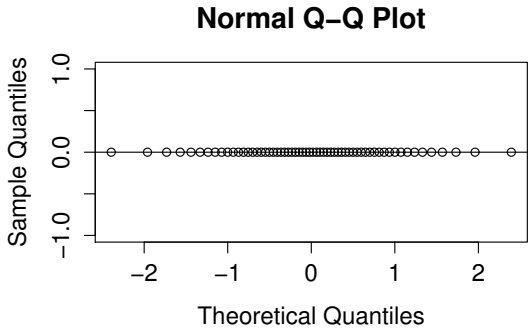
As stated earlier, we had planned on carrying out the analysis of variance test as of our experimental design but after evaluating our data, we noticed that all our data was non-normal and hence an ANOVA test could not be carried out. The assumption of normal data and homoscedasticity of variances would have been violated if we had continued with the ANOVA test. In light of that, we resolved to use a different test which was the Mann-Whitney-Wilcoxon test. This test had three assumptions [31] which were all fulfilled by our data. They include;

- The sample drawn from the population is random.
- Independence within the samples and mutual independence is assumed.
- Ordinal or numeric measurement scale is assumed.

The R function `wilcoxon.test()` was run at a 0.05 significance level with an alternative less to mean a one sided test. This gave the `P-Values` that were used to determine the statistical significance which is discussed in Chapter 7. The alternative less was used as from our alternative hypothesis we state that the mean coverage by CANoe is less than the mean coverage by CANoe$^+$. For each scenario we display the normality tests, Mann-Whitney-Wilcoxon test, the Q-Q plots with qqlines for the tools where applicable.

6.3.1 Without Faults

All Values for the failed tests in CANoe and CANoe$^+$ were identical hence the shapiro wilk normality test could not be described.



(a) Q-Q plot for CANoe

Figure 6.9: This is the Q-Q plot for failed testcases in CANoe and CANoe$^+$ without fault injections.

6.3.2 One Fault

A shapiro wilk test for normality was run for the Canoe and CANoe$^+$ tools and it confirmed that our data was non-normal. It was on this that we based to perform a Mann-Whitney-Wilcoxon non-parametric test to check if there was any statistical significance in our data.

	Significance Level	**W**	**P-Value**
Data	0.05	1641	0.1985

Table 6.4: Mann-Whitney-Wilcoxon table of results for the two tools with one fault injection.

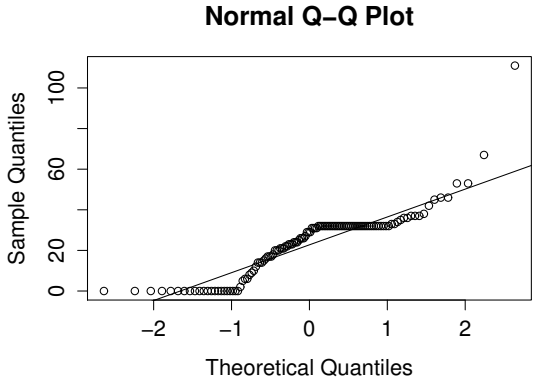


Figure 6.10: This is the Q-Q plot for failed testcases in CANoe and CANoe$^+$ with one fault injection.

6.3.3 Two Faults

The Canoe and CANoe$^+$ tools gave non normal data when the shapiro wilk test was run. Since we were analyzing two sets of data, a Mann-Whitney-Wilcoxon non-parametric test was run to determine if there was statistical significance between the two tools.

	Significance Level	**W**	**P-Value**
Data	0.05	580.5	6.503e-11

Table 6.5: Mann-Whitney-Wilcoxon table of results for the two tools with two fault injections.

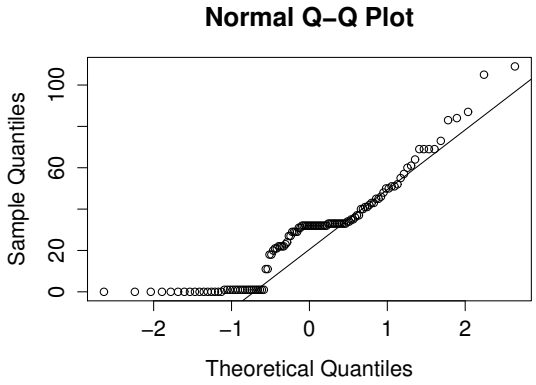


Figure 6.11: This is the Q-Q plot for failed testcases in CANoe and CANoe$^+$ with two fault injections.

6.3.4 Custom Fault

The shapiro wilk test for normality was only run for CANoe$^+$ and it showed non-normal data. The same test could not be run for CANoe as from the data, it was evident CANoe would never find the fault.

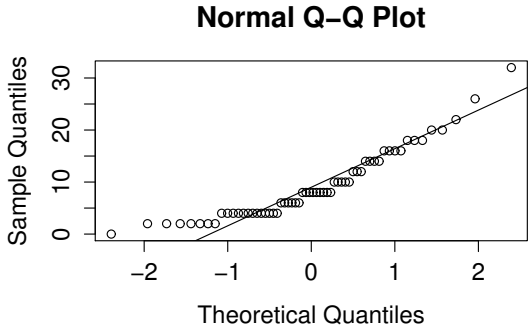


Figure 6.12: This is the Q-Q plot for failed testcases in CANoe$^+$ with a custom fault injection .

6.3.5 General Conclusion

In this section, we analyzed all the data we collected with respect to the tools in order to get an overview of the analysis.

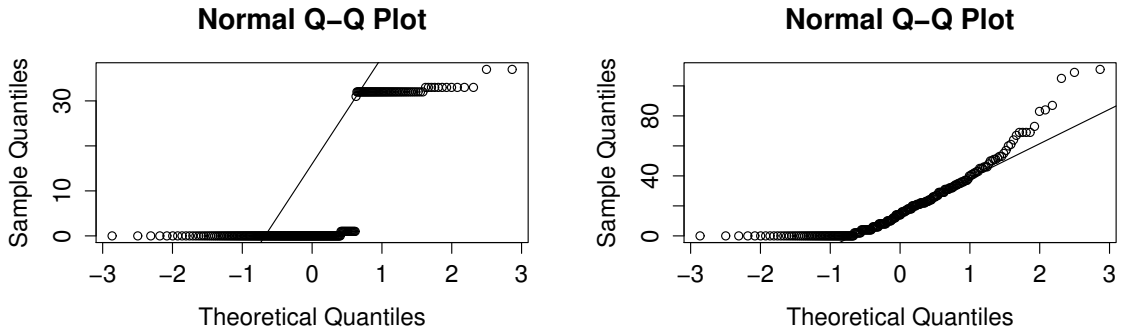
Method		
	CANoe	**CANoe$^+$**
Significance Level	0.05	0.05
W	0.5688	0.83857
P-Value	<2.2e-16	4.334e-15

(a) Shapiro wilk test for the two tools

	Significance Level	**W**	P-Value
Data	0.05	17750	1.003e-14

(b) Mann-Whitney-Wilcoxon table of results of the two tools for the general conclusion

Table 6.6: Shapiro wilk and Mann-Whitney-Wilcoxon test tables.



(a) Q-Q plot for CANoe

(b) Q-Q plot for CANoe$^+$

Figure 6.13: These are the Q-Q plots of failed testcases in CANoe and CANoe$^+$ for the general conclusion.

7
Discussion

7.1 Evaluation of Results and Implications

All the data was collected and analyzed according to five scenarios i.e. without faults, with one fault, two faults, custom fault and the general conclusion because the observations were different per scenario. The descriptive statistics for the five scenarios in which we run the experiment were analyzed to give both the central measures of tendency as well as the measures of dispersion for our data. Density plots were then used so as to show the distribution of the failed test cases with respect to the tool that was being used. Box plots were also used to show the shape of the data as well as the outliers in the data sets for each scenario.

The shapiro wilk tests were run for four scenarios with the exception of the without faults scenario at a significance level of 0.05 to see if the data was normally distributed or close to normal distribution. The without faults scenario was left out because all the collected data in this scenario was identical hence the test for normality could not be described. The null hypothesis for the shapiro wilk test was that the population was normally distributed. Thus if the `P-Value` was less than the chosen significance level of 0.05, then the null hypothesis would be rejected as that was enough evidence that the data tested was not from a normally distributed population hence the data was concluded to be non-normal. On the contrary, if the `P-Value` was greater than the significance level of 0.05, the null hypothesis could not be rejected.

From these tests, we were able to pass all the three assumptions for the Mann-Whitney-Wilcoxon test as our data was an independent random sample due to the random algorithms used in CANoe$^+$, a numeric scale was assumed and most importantly it was non-normal data. Lastly the Mann-Whitney-Wilcoxon tests were run to show statistical significance between the two tools for all the scenarios in accordance with the stated hypotheses.

For the scenarios where the difference in the mean coverage of the two tools was statistically significant, the effect sizes were calculated to show the size of the difference. A script for the calculation of non-parametric effect sizes which uses Vargha and Delaney's A statistic was used to calculate the effect sizes in each scenario where applicable [8].

The paper gives the following examples for interpreting the A-Statistic:
A ~0.56 => Small effect size (i.e. small superiority of X1 over X2)
A ~0.64 => Medium effect size (i.e. medium superiority of X1 over X2)
A ~0.71 => Large effect size (i.e. large superiority of X1 over X2)

Below is the analysis per scenario with the detailed fault injections where applicable;

7.1.1 Without faults

Like the heading suggests, for the first run, no faults were injected in the tested functionality and this was our baseline for the analysis while using the two tools. The experiment was executed with identical functionality and since it was the correct functionality for the sample functions, no faults were discovered hence reported faults were zero for both the tools. It was observed from the execution that without fault injections, the two tools had the same fault finding capabilities.

The `P-Values` for the shapiro wilk tests couldn't be described because there was zero variance. For the Mann-Whitney-Wilcoxon test, the `P-Value` was 1 and this meant that there was no difference in the mean coverage between the two tools: CANoe and CANoe$^+$.

7.1.2 One fault

The first random fault that was injected; a random function to set a random value to start the ignition. The random value is always either 1 or 0, where 1 starts the ignition and 0 stops the ignition. Depending on the random number generated; 1 will cause the ignition to start and 0 otherwise. It is worth mentioning that at least one feature is dependent on the ignition, if the ignition fails to start then executing the other functions that are dependant on it will also definitely fail.

It was observed from the execution of the experiment that whereas faults were only generated when a random value of 0 was set to the ignition, all the functions that depended on the ignition failed in CANoe$^+$ and there were no passed test cases at all. For the CANoe execution, passed test cases were recorded 30% of the time and this is not good for safety critical software.

By analyzing the shapiro wilk test for normality, it was seen that the `P-Values` were 4.177e-11 and 9.317e-07 for CANoe and CANoe$^+$ respectively at a 0.05 significance level. These values were less than the 0.05 significance level hence we reject the null hypothesis as there is enough evidence that the data is non-normal.

The `P-Value` from the Mann-Whitney-Wilcoxon test was 0.1985 which is greater than the significance level of 0.05, we conclude that there isn't a statistically significant difference in the mean coverage between the two tools: CANoe and CANoe$^+$.

7.1.3 Two faults

We extended the one fault scenario and injected a new random fault in the functionality of locking doors. The functionality for locking doors was not dependent on the ignition hence it could still function correctly even when the ignition had failed to start. The fault was similar to the fault injection in the one fault scenario; a random value was assigned to the locking functionality. The random number that gets assigned is either 0 or 1, where the bit 1 opens the door and is the normal behavior while bit 0 locks the door and causes the system under test to fail. Testing

this scenario was expected to report fails when assigning an invalid value to unlock the door, fail when the ignition had an invalid value, fail when both features were assigned invalid values, or pass if the random function generated the valid value.

From the executions observation of both the tools, in CANoe$^+$ there was no single test case that passed due to the injected faults which is how the tool should behave. But the CANoe tool managed to report passed test cases 26% of the time despite the faults.

The shapiro wilk test gave `P-Values` of CANoe and CANoe$^+$ as 2.014e-10 and 0.001159 respectively. Both these values were less than the 0.05 significance level hence depicted enough evidence to reject the null hypothesis and conclude that the two populations contained non-normal data. While analyzing the Mann-Whitney-Wilcoxon test, we observed that the `P-Value` was 6.503e-11 which is way below the 0.05 significance level for a one-tailed test hence there was enough evidence to reject our stated null hypothesis as there was a statistically significant difference in the mean coverage of the two tools.

We went ahead to calculate the effect size in order to know the size of the difference, and below is the result;

```
 Vargha and Delaney's A statistic, i.e. the measure of stochastic superiority

Call:
a.statistic.default(canoe = canoe, canoeplus = canoeplus)

A statistic = 0.16125
 i.e. the probability that a value from canoe is larger than a value from canoeplus is 16.12%

95% confidence interval for A = [0.103, 0.244]
 Difference is SIGNIFICANT at alpha = 0.05 level (Note: EXPERIMENTAL)

99% confidence interval for A = [0.089, 0.274]
 Difference is SIGNIFICANT at alpha = 0.01 level (Note: EXPERIMENTAL)

Superior: canoeplus
Effect size: Large
```

Figure 7.1: Effect size for the two faults injection using Vargha and Delaney's A-statistic.

From the effect size calculations, the A- statistic gives 0.16125 indicating a large effect size and that CANoe performed worse than CANoe$^+$. The values are interpreted as follows: 16.12% of the time CANoe will work better than CANoe$^+$. Equivalently, 83.88% of the time CANoe$^+$ will work better than CANoe.

7.1.4 Custom fault

A custom fault was injected in the door lock functionality that is responsible for unlocking the door. This fault would only be triggered if the unlocking feature of the door was executed two times successively, if any other action was executed between the clicks, the fault would not be triggered and the system under test would

execute normally with no errors. This fault sets the bit value of 0 when the fault is triggered else the bit 1.

This was the most interesting observation as in CANoe$^+$ the affected functionality of unlocking the door failed during all the runs but suprisingly CANoe passed 100% without a single fault reported. This can be attributed to the fact that CANoe executes sequentially hence is not designed to catch such faults. This kind of failure can be classified in the same way as the one that occured in the Ariane 5 rocket[15]. Such kinds of faults are hard to catch as they are not triggered in the usual execution and for safety critical systems like automotives, identifying them could be the difference between life and death.

The shapiro wilk test for CANoe was not applicable as there were 0 failed test cases but for CANoe$^+$ the `P-Value` was 0.0001187 which was less than the 0.05 significance level and sufficient evidence for the null hypothesis to be rejected which meant that the data from this scenario was non-normal. The Mann-Whitney-Wilcoxon test gave a `P-Value` of <2.2e-16 which was very much below the 0.05 significance level for a one tailed test. This was enough evidence to reject the null hypothesis as the highest statistical significant difference in the mean coverage of the tools was observed in this scenario.

The effect size for the scenario was calculated and below is the result;

```
 Vargha and Delaney's A statistic, i.e. the measure of stochastic superiority

Call:
a.statistic.default(canoe = canoe, canoeplus = canoeplus)

A statistic = 0.008333333
 i.e. the probability that a value from canoe is larger than a value from canoeplus is 0.83%

95% confidence interval for A = [0.001, 0.046]
 Difference is SIGNIFICANT at alpha = 0.05 level (Note: EXPERIMENTAL)

99% confidence interval for A = [0.001, 0.068]
 Difference is SIGNIFICANT at alpha = 0.01 level (Note: EXPERIMENTAL)

Superior: canoeplus
Effect size: Large
```

Figure 7.2: Effect size for the custom faults injection using Vargha and Delaney's A-statistic.

The effect size calculations, give the A- statistic as 0.008333333 indicating a large effect size and that CANoe performed worse than CANoe$^+$. The values are interpreted as follows: 0.83% of the time CANoe will work better than CANoe$^+$. Equivalently, 99.17% of the time CANoe$^+$ will work better than CANoe.

7.1.5 General conclusion

The last analysis that was conducted involved using all the data from the different scenarios to come up with one data set with respect to the tools. A shapiro wilk test was conducted and the `P-Values` for both the tools were less than 0.05 significance

level which signified non-normal data. We went on to perform the Mann-Whitney-Wilcoxon test and found that there was an overall statistical significance in the mean coverage as the `P-Value` was less than 0.05 significance level for a one tailed test.

With this data we were able to reject the null hypothesis and enforce our alternative hypothesis that indeed the mean coverage of the CANoe tool is less than the mean coverage of the developed prototype of CANoe$^+$. From this we can confidently affirm that this study positively confirmed the theory that the use of CANoe$^+$ in automotive systems increases coverage of test cases as compared to the use of CANoe.

```
 Vargha and Delaney's A statistic, i.e. the measure of stochastic superiority

Call:
a.statistic.default(canoe = canoe, canoeplus = canoeplus)

A statistic = 0.3081684
 i.e. the probability that a value from canoe is larger than a value from canoeplus is 30.82%

95% confidence interval for A = [0.264, 0.356]
 Difference is SIGNIFICANT at alpha = 0.05 level (Note: EXPERIMENTAL)

99% confidence interval for A = [0.251, 0.372]
 Difference is SIGNIFICANT at alpha = 0.01 level (Note: EXPERIMENTAL)

Superior: canoeplus
Effect size: Large
```

Figure 7.3: Effect size for the general conclusion using Vargha and Delaney's A-statistic.

The effect size calculations, give the A- statistic as 0.3081684 indicating a large effect size and that CANoe performed worse than CANoe$^+$. The values are interpreted as follows: 30.82% of the time CANoe will work better than CANoe$^+$. Equivalently, 69.18% of the time CANoe$^+$ will work better than CANoe.

7.2 Inferences

From our findings, we believe that the use of model-based testing in the automotive industry facilitates rapid exercising of the software under test to discover faults that can't normally be unmasked using the usual testing methods. Our solution is not limited to CANoe and the automotive industry, any test module based on .NET/C# can benifit from our solution. More so, this model-based testing approach can be extended to more safety critical systems like avionics to ensure that there is no hidden undesirable functionality that can cause failures.

As had been stated earlier, we had planned on using the NASA Task Load Index to report on the effort needed in terms of efficiency to use our developed prototype but since we were the subjects of the experiment the result would have been biased had we filled it in. We believe that CANoe$^+$ does not take alot of effort in terms of mental, physical and temporal demand. However we leave it to future studies to use the prototype and assess the workload.

Due to the nature of the approach, regression testing is also made possible as whenever new functionality is added, the whole test suite can be exercised over and again with limited effort.

However, it has to be noted that for the approach to be effective, it has to be done on a specific function basis i.e. it would be hard to build one model for a whole system to be used for model-based testing. It is better if a module to be tested is identified, modelled and then tested with the model-based testing approach.

8

Threats to Validity

In this section we discuss all the threats that could have had an impact on the validity of our results and how they were mitigated [37].

8.1 Conclusion Validity

Violated assumptions of statistical tests could have been a threat but before concluding to use the Mann-Whitney-Wilcoxon test, we made sure that all the assumptions of the test had not been violated i.e. the collected data was an independent random sample, a numeric scale was assumed and most importantly it was non-normal data.

Fishing for a specific result, could have been a threat but both the CANoe and CANoe$^+$ tools were automatically executed to assess the fault finding capability for each of the tools. From the execution, the failed test cases from each tool were noted down and there is no possible way in which a specific outcome could have been fished.

The sample software was executed for a total of 480 runs for the two tools to mitigate the threat of measurements reliability. The total failed test cases were recorded from the automatic execution of the tools. The runs and injected faults were considered enough as adding faults would just diverge the results more to show that CANoe$^+$ is a better treatment.

Since the authors were the subjects in the experiment, the reliability of treatment implementation was reduced by having the subjects perform the same exact actions during the experiment. Hence, the treatment was as standard as possible over the subjects and occasions.

8.2 Internal Validity

There was no perceived threat to the internal validity. The design of the CANoe$^+$ tool was in such a way that it used random algorithms which mandated the elimination of human subjects and instead promoted the involvement of the authors to execute the software multiple times so as to find faults.

Each of the authors(subjects) would apply a treatment to the sample software successively e.g. when executing the scenario where no faults were inserted, the subject would execute the software with CANoe then CANoe$^+$ interchangeably during the experiment hence ruling out the history threat.

Maturation and testing couldn't have been threats since the experiment involved automatic execution of the sample software, and all the subjects had to do was to

note down the test metrics.

A nested design was used hence we had one level for the coverage of the generated tests and two levels for the tool usage as shown in Figure 5.1. Within the nested design, a crossed design was used to randomly assign the subjects to the treatments to cross out the variability that could have affected the experiment results. By so doing we mitigated the threats to internal validity which are concerned with the influences that can affect the independent variable with respect to causality, without the researcher's knowledge.

8.3 Construct Validity

The interaction of testing and treatment threat was tackled by having well defined faults prior to the actual process of injecting faults. This ensured uniformity of the tests and the subjects were under no influence to become either sensitive or receptive to the treatment.

To avoid unintended negative constructs like experience of the subjects in the experiment, the experience of the subjects was used as a blocked variable hence focusing on only the fault finding capability of the two tools.

Besides the above, other threats were amicably mitigated as noted below;

To prevent experimenter expectancies or bias, random algorithms were used in CANoe$^+$ and the execution of the two tools for failed test cases was fully automated.

For inadequate preoperational explication of constructs, our constructs were evidently defined as well as the measures i.e. evaluating the effectiveness of each of the tools at catching faults and this was translated into the number of failed test cases for each tool.

Interraction of different treatments threat was minimized due to the automatic execution of the two tools and there was no way in which the effect of the two treatments was due to a combination of the treatments. This applied to hypothesis guessing as well.

8.4 External Validity

There were no observed external validity threats. For interaction of selection and treatment, the authors who are both software developers and testers participated in the experiment and this was a good representation of the population we wanted to generalize to. These tools are mostly used by software developers and software testers in the industry.

To rule out the interaction of setting and treatment validity threat, the same exact tools that are used in the industry i.e. CANoe 8, Visual Studio 2012 were used during the execution on the two tools in addition to Eclipse which was used for CANoe$^+$.

All in all, we aimed to make the execution environment as realistic as possible so as to increase the generalizability of our prototype.

9
Conclusions

Our contribution in this thesis was the prototype for our tool CANoe$^+$ which was developed with the aim to investigate if there is coverage maximization of model-based test cases with our new tool(CANoe$^+$) as compared to the current way of working which is CANoe. Sample functions were developed e.g. closing and opening of a car door, starting and stopping the ignition, rolling up and down the windows and turning on and off of lights in a car system that was simulated in the CANoe software.

The sample functions were then executed with the use of the two tools to determine which was best at catching faults. This fault finding capability was translated into the coverage. The functions were executed multiple times i.e. 480 runs while injecting faults to assess the fault finding capabilities for each of the tools. For each run, the number of failed test cases were recorded and later used in the analysis. The collected data was statistically analyzed and reported in form of a controlled experiment.

During the analysis, the Mann-Whitney-Wilcoxon one-tailed test was used at an alpha level of 0.05. We were able to reject the null hypothesis in favor of the alternative hypothesis as the results reinforced the superiority of model-based testing approaches like CANoe$^+$ over testing methods like CANoe. The results gave enough evidence for us to affirm that the use of CANoe$^+$ in automotive systems increases the coverage of test cases as compared to the use of CANoe.

The limitation to this approach is that for it to be effective, functions that need to be tested have to be identified, modelled and then tested as opposed to modelling the whole system as a whole.

Future Work

To ascertain that, this solution is not limited to CANoe or the automotive industry, it needs to be tried out in other fields let alone other projects and also the work load assessed using the NASA Task Load Index in Appendix B.

Bibliography

[1] H. Altinger, F. Wotawa, and M. Schurius. Testing methods used in the automotive industry: Results from a survey. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 1–6. ACM, 2014.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10, May 2011.

[3] F. Belli, A. Hollmann, and M. Kleinselbeck. A graph-model-based testing method compared with the classification tree method for test case generation. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 193–200, July 2009.

[4] E. Bringmann and A. Kramer. Model-based testing of automotive systems. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 485–493, April 2008.

[5] M. R. V. Chaudron, W. Heijstek, and A. Nugroho. How effective is uml modeling ? *Software & Systems Modeling*, 11(4):571–580, 2012.

[6] M. Conrad, I. Fey, and S. Sadeghipour. Systematic model-based testing of embedded automotive software. *Electronic Notes in Theoretical Computer Science*, 111:13–26, 2005.

[7] W. Dulz and F. Zhen. Matelo-statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 336–342. IEEE, 2003.

[8] R. Feldt. Non-parametric effect size calculations. *http://www.cse.chalmers.se/~feldt/advice/statistics/nonparametric_effect_sizes.r*, 31 May. 2016.

[9] D. Fodor and K. Enisz. Vehicle dynamics based abs ecu verification on real-time hardware-in-the-loop simulator. In *Power Electronics and Motion Control Conference and Exposition (PEMC), 2014 16th International*, pages 1247–1251, Sept 2014.

[10] G. Frey, R. Drath, B. Schlich, and R. Eschbach. A new specification language for the development of plc safety applications. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–8, Sept 2012.

[11] Graphwalker.org. Home | graphwalker the open source model-based testing tool. *http://graphwalker.org/index*, 16 Dec. 2015.

[12] V. Gudmundsson, M. Lindvall, L. Aceto, J. Bergthorsson, and D. Ganesan. Model-based testing of mobile systems–an empirical study on quizup android

app.
[13] A. Guiotto, B. Acquaroli, and A. Martelli. Matelo: automated testing suite for software validation. In *DASIA 2003*, volume 532, page 30, 2003.
[14] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proceedings of the 48th Design Automation Conference*, pages 224–229. ACM, 2011.
[15] ima.umn.edu/ arnold/disasters/ariane.html. The explosion of the ariane 5. *https://www.ima.umn.edu/~arnold/disasters/ariane.html*, 15 May. 2016.
[16] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer, 2008.
[17] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey. Model-based testing of embedded automotive software using mtest. In *SAE World Congress*, pages 8–11. Citeseer, 2004.
[18] B. Marculescu, S. Poulding, R. Feldt, K. Petersen, and R. Torkar. Tester interactivity makes a difference in search-based software testing: A controlled experiment. *arXiv preprint arXiv:1512.04812*, 2015.
[19] M. Mofidpoor, N. Shiri, and T. Radhakrishnan. Index-based join operations in hive. In *Big Data, 2013 IEEE International Conference on*, pages 26–33, Oct 2013.
[20] C. Murphy, Z. Zoomkawalla, and K. Narita. Automatic test case generation and test suite reduction for closed-loop controller software. 2013.
[21] D.-I. F. N. Niedermark, F. Löw, and D.-I. S. Müller. Automated data-driven validation of the diagnostic implementation. *ATZelektronik worldwide*, 10(6):22–25, 2015.
[22] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh. An integrated test generation tool for enhanced coverage of simulink/stateflow models. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 308–311, March 2012.
[23] P. Peti, A. Timmerberg, T. Pfeffer, S. Muller, C. Ratz, and V. Informatik. A quantitative study on automatic validation of the diagnostic services of electronic control units. In *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pages 799–808, Sept 2008.
[24] S. L. Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1(1):219–253, 1995.
[25] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.
[26] H. Robinson. Graph theory techniques in model-based testing. In *International Conference on Testing Computer Software*, volume 1, page 999, 1999.
[27] S. Rosaria and H. Robinson. Applying models in your testing process. *Information and Software technology*, 42(12):815–824, 2000.
[28] S. Siegl, W. Dulz, R. German, and G. Kiffe. Model-driven testing based on markov chain usage models in the automotive domain. In *12th European Work-*

shop on Dependable Computing, EWDC 2009, pages 6–pages, 2009.

[29] S. Sivanandan and Y. C. B. Agile development cycle: Approach to design an effective model based testing with behaviour driven automation framework. In *Advanced Computing and Communications (ADCOM), 2014 20th Annual International Conference on*, pages 22–25, Sept 2014.

[30] K.-P. Spring, T. Hohmann, and K. Hahmann. Porsche validates gateway ecus automatically. *ATZelektronik worldwide*, 5(6):46–49.

[31] statisticssolutions.com. Mann-whitney u test - statistics solutions. *http://www.statisticssolutions.com/mann-whitney-u-test/*, 15 May. 2016.

[32] T. Tamisier and F. Feltz. *Digital Information and Communication Technology and Its Applications: International Conference, DICTAP 2011, Dijon, France, June 21-23, 2011, Proceedings, Part II*, chapter Lifelong Automated Testing: A Collaborative Framework for Checking Industrial Products Along Their Lifecycle, pages 80–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[33] Vector.com. Canoe - ecu development and test. *http://vector.com/vi_canoe_en.html*, 16 Dec. 2015.

[34] Vector.com. Nasa tlx paper/pencil version. *http://humansystems.arc.nasa.gov/groups/tlx/paperpencil.html*, 25 Apr. 2016.

[35] wikipedia.com. Electronic control unit - wikipedia, the free encyclopedia. *https://en.wikipedia.org/wiki/Electronic_control_unit*, 21 apr. 2015.

[36] C. Wohlin, M. Höst, and K. Henningsson. Empirical research methods in web and software engineering. In *Web engineering*, pages 409–430. Springer, 2006.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[38] X. Yang, Y. Lin, F. Gao, G. Wang, and Y. Zhang. Automated test system design of body control module. In *Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on*, volume 3, pages 1542–1546, April 2014.

[39] yworks.com. vteststudio - test authoring tool for embedded systems. *http://vector.com/vi_vteststudio_en.html*, 06 May. 2016.

[40] yworks.com. yed - graph editor. *https://www.yworks.com/products/yed/gallery*, 15 April. 2016.

A

Appendix 1

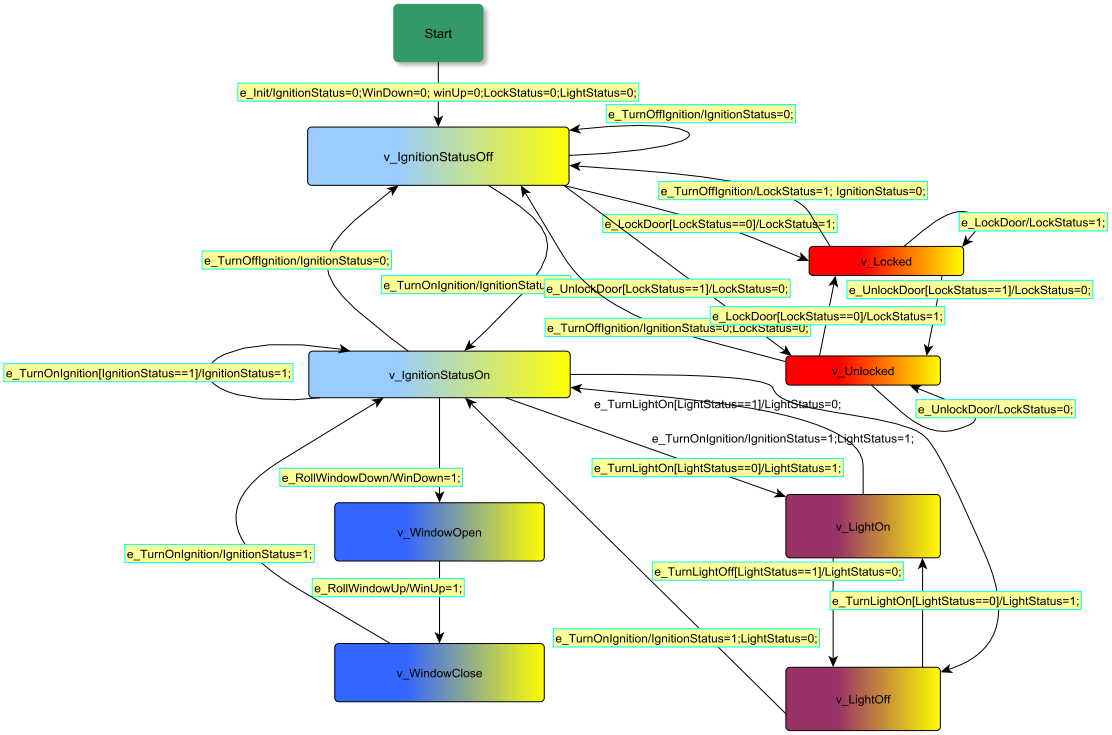


Figure A.1: Sample model for the four modelled functionalities i.e. closing and opening of a car door, starting and stopping the ignition, rolling up and down the windows and turning on and off of lights in a car system

B

Appendix 2

NASA Task Load Index

Hart and Staveland's NASA Task Load Index (TLX) method assesses work load on five 7-point scales. Increments of high, medium and low estimates for each point result in 21 gradations on the scales.

Name	Task	Date

Mental Demand — How mentally demanding was the task?

Very Low — Very High

Physical Demand — How physically demanding was the task?

Very Low — Very High

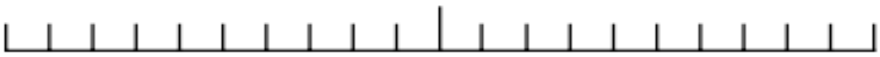
Temporal Demand — How hurried or rushed was the pace of the task?

Very Low — Very High

Performance — How successful were you in accomplishing what you were asked to do?

Perfect — Failure

Effort — How hard did you have to work to accomplish your level of performance?



Very Low — Very High

Frustration — How insecure, discouraged, irritated, stressed, and annoyed wereyou?

Very Low — Very High

C

Appendix 3

Total number of test runs: 60									
		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	1	37	37	0	00:00:29	187	187	0	00:05:48
	2	37	37	0	00:00:30	80	80	0	00:02:33
	3	37	37	0	00:00:29	102	102	0	00:03:07
	4	37	37	0	00:00:30	142	142	0	00:04:20
	5	37	37	0	00:00:29	106	106	0	00:03:11
	6	37	37	0	00:00:29	193	193	0	00:05:46
	7	37	37	0	00:00:29	120	120	0	00:03:35
	8	37	37	0	00:00:29	179	179	0	00:05:20
	9	37	37	0	00:00:30	154	154	0	00:04:35
	10	37	37	0	00:00:29	172	172	0	00:04:58
	11	37	37	0	00:00:30	98	98	0	00:02:57
	12	37	37	0	00:00:30	329	329	0	00:09:14
	13	37	37	0	00:00:29	241	241	0	00:06:59
	14	37	37	0	00:00:29	340	340	0	00:09:43
	15	37	37	0	00:00:30	128	128	0	00:03:51
	16	37	37	0	00:00:29	165	165	0	00:04:51
	17	37	37	0	00:00:30	198	198	0	00:05:46
	18	37	37	0	00:00:30	108	108	0	00:03:14
	19	37	37	0	00:00:29	249	249	0	00:07:05
	20	37	37	0	00:00:30	193	193	0	00:05:36
	21	37	37	0	00:00:30	186	186	0	00:05:27
	22	37	37	0	00:00:29	200	200	0	00:05:51
	23	37	37	0	00:00:29	189	189	0	00:05:28
	24	37	37	0	00:00:30	154	154	0	00:04:35
	25	37	37	0	00:00:30	236	236	0	00:07:13
	26	37	37	0	00:00:30	263	263	0	00:07:30
	27	37	37	0	00:00:29	229	229	0	00:06:30
	28	37	37	0	00:00:29	211	211	0	00:06:02
	29	37	37	0	00:00:29	317	317	0	00:08:59
Without faults	30	37	37	0	00:00:30	529	529	0	00:14:42

Figure C.1: Collected data for the two tools without faults, continued on next page

		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	31	37	37	0	00:00:29	329	329	0	00:09
	32	37	37	0	00:00:30	138	138	0	00:04
	33	37	37	0	00:00:30	117	117	0	00:03:34
	34	37	37	0	00:00:29	134	134	0	00:04:24
	35	37	37	0	00:00:30	198	198	0	00:05:51
	36	37	37	0	00:00:29	454	454	0	00:12:48
	37	37	37	0	00:00:30	223	223	0	00:06:34
	38	37	37	0	00:00:30	207	207	0	00:06:04
	39	37	37	0	00:00:29	223	223	0	00:06:34
	40	37	37	0	00:00:29	168	168	0	00:05:02
	41	37	37	0	00:00:29	162	162	0	00:04:49
	42	37	37	0	00:00:29	224	224	0	00:06:27
	43	37	37	0	00:00:29	138	138	0	00:04:12
	44	37	37	0	00:00:30	223	223	0	00:06:23
	45	37	37	0	00:00:29	173	173	0	00:05:01
	46	37	37	0	00:00:30	135	135	0	00:04:04
	47	37	37	0	00:00:30	216	216	0	00:06:24
	48	37	37	0	00:00:30	276	276	0	00:07:59
	49	37	37	0	00:00:30	226	226	0	00:06:41
	50	37	37	0	00:00:30	258	258	0	00:07:28
	51	37	37	0	00:00:30	134	134	0	00:04:09
	52	37	37	0	00:00:29	131	131	0	00:04:04
	53	37	37	0	00:00:29	115	115	0	00:03:30
	54	37	37	0	00:00:30	279	279	0	00:07:59
	55	37	37	0	00:00:29	206	206	0	00:05:59
	56	37	37	0	00:00:29	561	561	0	00:15:33
	57	37	37	0	00:00:29	271	271	0	00:07:49
	58	37	37	0	00:00:30	232	232	0	00:06:45
	59	37	37	0	00:00:30	178	178	0	00:05:16
Without faults	60	37	37	0	00:00:29	139	139	0	00:04:08

Figure C.2: Collected data for the two tools without faults

Total number of test runs: 60									
		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	1	37	5	32	00:00:30	240	194	46	00:07
	2	37	37	0	00:00:30	180	157	23	00:05
	3	37	37	0	00:00:29	114	109	5	00:03:27
	4	37	5	32	00:00:30	171	145	26	00:05:03
	5	37	5	32	00:00:30	82	74	8	00:02:34
	6	37	5	32	00:00:30	127	113	14	00:03:48
	7	37	37	0	00:00:30	168	153	15	00:04:59
	8	37	5	32	00:00:30	172	148	24	00:05:07
	9	37	37	0	00:00:30	157	141	16	00:04:44
	10	37	37	0	00:00:30	202	177	25	00:05:55
	11	37	5	32	00:00:30	182	161	21	00:05:23
	12	37	37	0	00:00:30	132	108	24	00:04:02
	13	37	5	32	00:00:30	151	120	31	00:04:28
	14	37	5	32	00:00:30	171	157	14	00:05:03
	15	37	5	32	00:00:30	132	99	33	00:04:04
	16	37	5	32	00:00:30	165	139	26	00:04:51
	17	37	5	32	00:00:30	285	232	53	00:08:14
	18	37	37	0	00:00:29	208	188	20	00:06:07
	19	37	5	32	00:00:29	96	84	12	00:02:58
	20	37	37	0	00:00:29	191	158	33	00:05:41
	21	37	37	0	00:00:29	163	134	29	00:04:51
	22	37	37	0	00:00:30	235	213	22	00:06:54
	23	37	37	0	00:00:29	142	125	17	00:04:14
	24	37	5	32	00:00:30	174	152	22	00:05:02
	25	37	37	0	00:00:30	128	110	18	00:03:46
	26	37	5	32	00:00:30	198	181	17	00:05:47
	27	37	37	0	00:00:30	89	83	6	00:02:46
	28	37	37	0	00:00:30	281	236	45	00:08:04
	29	37	5	32	00:00:30	259	228	31	00:07:26
With 1 fault	30	37	37	0	00:00:29	132	108	24	00:03:58

Figure C.3: Collected data for the two tools with one fault injection, continued on next page

		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	31	37	5	32	00:00:30	267	238	29	00:07
	32	37	5	32	00:00:30	200	174	26	00:05
	33	37	5	32	00:00:30	279	242	37	00:08:09
	34	37	37	0	00:00:29	250	236	14	00:07:21
	35	37	37	0	00:00:30	206	175	31	00:06:04
	36	37	5	32	00:00:30	326	299	27	00:09:25
	37	37	5	32	00:00:30	244	207	37	00:07:04
	38	37	5	32	00:00:30	202	171	31	00:06:03
	39	37	37	0	00:00:30	221	168	53	00:06:34
	40	37	5	32	00:00:30	498	387	111	00:14:05
	41	37	5	32	00:00:30	189	160	29	00:05:50
	42	37	5	32	00:00:30	173	152	21	00:05:24
	43	37	37	0	00:00:29	124	104	20	00:03:52
	44	37	5	32	00:00:29	150	141	9	00:04:31
	45	37	37	0	00:00:30	232	198	34	00:06:59
	46	37	37	0	00:00:30	382	344	38	00:11:02
	47	37	5	32	00:00:30	97	91	6	00:03:02
	48	37	5	32	00:00:30	314	278	36	00:18:48
	49	37	5	32	00:00:30	184	161	23	00:05:27
	50	37	37	0	00:00:29	189	187	2	00:05:41
	51	37	5	32	00:00:30	346	304	42	00:10:05
	52	37	5	32	00:00:30	263	228	35	00:07:43
	53	37	5	32	00:00:29	147	127	20	00:04:28
	54	37	5	32	00:00:30	210	174	36	00:06:07
	55	37	5	32	00:00:30	112	102	10	00:03:33
	56	37	5	32	00:00:30	387	320	67	00:11:08
	57	37	5	32	00:00:30	154	133	21	00:04:43
	58	37	5	32	00:00:30	368	322	46	00:10:33
	59	37	5	32	00:00:30	134	117	17	00:04:07
With 1 fault	60	37	37	37	00:00:30	162	139	23	00:04:54

Figure C.4: Collected data for the two tools with one fault injection

Total number of test runs: 60									
		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	1	37	36	1	00:00:30	224	181	43	00:06
	2	37	37	0	00:00:29	233	181	52	00:06
	3	37	5	32	00:00:30	237	186	51	00:06:55
	4	37	36	1	00:00:30	162	131	31	00:04:48
	5	37	36	1	00:00:30	223	182	41	00:06:40
	6	37	4	33	00:00:31	105	94	11	00:03:22
	7	37	5	32	00:00:30	128	110	18	00:03:54
	8	37	37	0	00:00:30	172	145	27	00:05:09
	9	37	37	0	00:00:29	84	73	11	00:02:37
	10	37	36	1	00:00:30	102	84	18	00:03:06
	11	37	5	32	00:00:30	390	321	69	00:12:47
	12	37	5	32	00:00:30	203	134	69	00:05:58
	13	37	5	32	00:00:30	161	129	32	00:04:47
	14	37	36	1	00:00:30	98	76	22	00:03:08
	15	37	36	1	00:00:30	175	151	24	00:05:11
	16	37	36	1	00:00:30	127	98	29	00:03:51
	17	37	37	0	00:00:29	277	208	69	00:08:10
	18	37	36	1	00:00:30	299	216	83	00:08:36
	19	37	4	33	00:00:30	148	105	43	00:04:27
	20	37	37	0	00:00:29	143	98	45	00:04:21
	21	37	37	0	00:00:29	170	135	35	00:05:03
	22	37	36	1	00:00:30	164	128	36	00:04:52
	23	37	36	1	00:00:30	161	140	21	00:03:31
	24	37	4	33	00:00:30	460	355	105	00:13:01
	25	37	4	33	00:00:30	186	126	60	00:05:32
	26	37	4	33	00:00:31	186	129	57	00:05:24
	27	37	36	1	00:00:30	223	177	46	00:06:26
	28	37	37	0	00:00:30	209	169	40	00:06:20
	29	37	37	0	00:00:30	291	204	87	00:08:23
With 2 Random faults	30	37	5	32	00:00:30	171	110	61	00:05

Figure C.5: Collected data for the two tools with two random faults Injected, continued on the next page

		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	31	37	37	0	00:00:29	253	203	50	00:07
	32	37	37	0	00:00:29	124	102	22	00:03
	33	37	37	0	00:00:29	148	126	22	00:04:31
	34	37	5	32	00:00:30	273	251	22	00:07:53
	35	37	36	1	00:00:30	186	138	48	00:05:32
	36	37	4	33	00:00:31	124	103	21	00:03:53
	37	37	5	32	00:00:30	112	92	20	00:03:34
	38	37	36	1	00:00:30	150	127	23	00:04:29
	39	37	5	32	00:00:30	146	104	42	00:04:22
	40	37	0	37	00:00:30	130	98	32	00:03:58
	41	37	37	0	00:00:30	113	79	34	00:03:34
	42	37	6	31	00:00:31	150	113	37	00:04:31
	43	37	5	32	00:00:30	106	77	29	00:03:21
	44	37	4	33	00:00:30	195	150	45	00:05:47
	45	37	36	1	00:00:30	240	167	73	00:07:00
	46	37	4	33	00:00:31	246	177	69	00:07:19
	47	37	5	32	00:00:30	233	149	84	00:07:02
	48	37	37	0	00:00:29	113	73	40	00:03:35
	49	37	36	1	00:00:30	123	89	34	00:03:48
	50	37	37	0	00:00:29	138	109	29	00:04:11
	51	37	36	1	00:00:30	269	205	64	00:07:57
	52	37	36	1	00:00:30	114	85	29	00:03:29
	53	37	5	32	00:00:30	190	139	51	00:05:46
	54	37	5	32	00:00:30	486	377	109	00:13:59
	55	37	4	33	00:00:31	113	58	55	00:03:35
	56	37	4	33	00:00:31	180	130	50	00:05:27
	57	37	37	0	00:00:30	143	111	32	00:04:26
	58	37	36	1	00:00:30	182	141	41	00:05:32
	59	37	4	33	00:00:31	141	114	27	00:04:17
With 2 Random faults	60	37	37	0	00:00:29	126	91	35	00:03:54

Figure C.6: Collected data for the two tools with two random faults Injected

Total number of test runs: 60									
		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	1	37	37	0	00:00:30	223	215	8	00:06
	2	37	37	0	00:00:30	189	171	18	00:05
	3	37	37	0	00:00:30	208	200	8	00:05:53
	4	37	37	0	00:00:29	292	278	14	00:08:46
	5	37	37	0	00:00:30	377	361	16	00:10:39
	6	37	37	0	00:00:29	154	142	12	00:04:46
	7	37	37	0	00:00:30	226	218	8	00:06:24
	8	37	37	0	00:00:30	229	213	16	00:06:34
	9	37	37	0	00:00:30	362	352	10	00:10:06
	10	37	37	0	00:00:30	156	142	14	00:04:33
	11	37	37	0	00:00:29	193	173	20	00:05:35
	12	37	37	0	00:00:29	206	184	22	00:05:58
	13	37	37	0	00:00:30	196	194	2	00:05:35
	14	37	37	0	00:00:29	225	213	12	00:06:22
	15	37	37	0	00:00:29	139	131	8	00:04:14
	16	37	37	0	00:00:29	106	104	2	00:03:08
	17	37	37	0	00:00:30	464	448	16	00:12:48
	18	37	37	0	00:00:29	209	195	14	00:05:59
	19	37	37	0	00:00:30	266	248	18	00:07:30
	20	37	37	0	00:00:30	85	77	8	00:02:40
	21	37	37	0	00:00:30	231	221	10	00:06:29
	22	37	37	0	00:00:29	256	224	32	00:07:19
	23	37	37	0	00:00:30	176	170	6	00:05:01
	24	37	37	0	00:00:29	176	174	2	00:05:07
	25	37	37	0	00:00:29	145	139	6	00:04:27
	26	37	37	0	00:00:30	195	191	4	00:05:38
	27	37	37	0	00:00:30	210	206	4	00:05:56
	28	37	37	0	00:00:30	238	230	8	00:06:43
	29	37	37	0	00:00:30	282	256	26	00:08:09
Custom fault	30	37	37	0	00:00:29	222	214	8	00:06:17

Figure C.7: Collected data for the two tools with a custom fault Injected, continued on the next page

		CANoe				CANoe+			
		Total	Passed	Failed	Time	Total	passed	Failed	Time
	31	37	37	0	00:00:29	210	206	4	00:06
	32	37	37	0	00:00:29	261	257	4	00:07
	33	37	37	0	00:00:29	199	185	14	00:05:50
	34	37	37	0	00:00:30	198	194	4	00:05:45
	35	37	37	0	00:00:30	189	187	2	00:05:23
	36	37	37	0	00:00:29	214	196	18	00:06:14
	37	37	37	0	00:00:30	204	194	10	00:05:56
	38	37	37	0	00:00:29	211	195	16	00:06:07
	39	37	37	0	00:00:29	102	96	6	00:03:07
	40	37	37	0	00:00:30	113	107	6	00:03:55
	41	37	37	0	00:00:30	112	106	6	00:03:27
	42	37	37	0	00:00:30	266	262	4	00:07:25
	43	37	37	0	00:00:29	373	365	8	00:10:28
	44	37	37	0	00:00:29	144	144	0	00:04:13
	45	37	37	0	00:00:29	120	108	12	00:03:41
	46	37	37	0	00:00:29	415	395	20	00:11:39
	47	37	37	0	00:00:30	429	427	2	00:11:54
	48	37	37	0	00:00:30	115	105	10	00:03:32
	49	37	37	0	00:00:29	395	391	4	00:10:53
	50	37	37	0	00:00:29	370	366	4	00:10:14
	51	37	37	0	00:00:29	119	115	4	00:03:33
	52	37	37	0	00:00:29	217	207	10	00:06:14
	53	37	37	0	00:00:30	339	337	2	00:09:16
	54	37	37	0	00:00:29	204	196	8	00:05:55
	55	37	37	0	00:00:30	168	162	6	00:04:51
	56	37	37	0	00:00:30	412	408	4	00:11:22
	57	37	37	0	00:00:29	131	127	4	00:03:52
	58	37	37	0	00:00:30	201	199	2	00:05:43
	59	37	37	0	00:00:30	163	159	4	00:04:44
Custom Fault	60	37	37	0	00:00:29	398	394	4	00:11:06

Figure C.8: Custom Fault Injection