Standard-Compliant Testing for Safety-Related Automotive Software

Anila Mjeda

This thesis is submitted to University of Limerick in fulfilment of the requirements for the degree of Doctor of Philosophy

Department of Computer Science and Information Systems

Supervisor: Professor Mike Hinchey

Submitted to University of Limerick, November 2013
To my mother Matilde Mjeda (née Gjonej) gone now but never forgotten, and, to my daughter Amy Isabelle Muçko. The two women in my life that only met in my heart, and both, shaped me profoundly.
Abstract

Standard-Compliant Testing for Safety-Related Automotive Software

Anila Mjeda

This thesis proposes a new testing method called CTM\textsubscript{CONTROL}, which enhances the popular Classification Tree Method (CTM) by incorporating Modified Condition/Decision Coverage (MC/DC) capabilities. This brings the CTM closer to the quality assurance requirements of the latest standards that regulate automotive software.

CTM\textsubscript{CONTROL} has a particular focus on the specification-based control-logic of the system under test (SUT). It incorporates MC/DC coverage of the logical expressions guarding the transitions of a system represented via Statecharts and it allows testing of the dependencies among the input values and the control logic of the SUT. The experimental results reported in this thesis point to the fact that CTM\textsubscript{CONTROL} can capture a group of errors in the control-logic of the SUT, which would not be captured via the classic CTM approach. The additional group of errors captured by CTM\textsubscript{CONTROL}, includes errors which can cause unwanted activations or unwanted feature interactions in a system. CTM\textsubscript{CONTROL} has been prototypically implemented in the MATLAB/Simulink/Stateflow environment.

This thesis also proposes the Formalization of a Reactive Testing Process (FRTP) which builds a pathway between reactive testing and the field of formal testing. The FRTP defines a number of ordered steps to be followed in a reactive testing process and a flow of logic on how to move between them. The steps serve as methodological pointers to help with the integration of formality in the automotive software testing process.

The FRTP includes a new Taxonomy of Reactive Testing (TRT). The TRT helps the test designer to elucidate the reactivity dimensions which are of interest for a specific testing process. To facilitate the use of formalization in industry, examples specified in Z notation are given for each of the dimensions of the TRT. The examples are meant as informational models, which can guide the “\textit{typical automotive software tester}” in the formalization of test reactivity. Also, changes are proposed to the latest model-based testing taxonomy, via incorporating test reactivity as one of the test-selection criteria. This accounts for those reactive test processes where test generation is performed on the fly.

All of the contributions of this research are tailored to the field of safety-related automotive software.
Acknowledgements

First of all I want to express my deeply-felt thanks and gratitude to my supervisor Professor Mike Hinchey for the invaluable suggestions, guidance, expert knowledge and the sheer pleasure of witnessing his intellect at work.

Also my thanks and gratitude go to my previous supervisors Professor Kevin Ryan and Dr Gabriel Leen. I am indebted to both of them for their advice, guidance and generosity with their time.

I would like to thank my examiners Dr Norah Power and Professor Jonathan P. Bowen for examining this thesis and for their many constructive comments.

My colleagues in Lero have provided much needed buoying with their support, friendship and humour. In particular, I would like to thank Hans-Jürgen Kugler and Prof. Stefan Kowalewski for their support and expert knowledge when this research was at its beginnings.

My husband Tani has been wonderful in his unconditional love and support. Foremost, I thank him for his patience to endure a PhD by proxy. My daughter Amy has had to give up a lot of time with mummy. I thank her for bringing so much joy, for being her wonderful self and for loving me “to the moon and back”. My sister Silvana has been an irreplaceable support in many more ways that she knows. She is the best sister and friend anyone could wish for. My gratitude also goes to Magdalena Varfi for her unselfish support and her willingness to travel thousands of miles to help her daughter-in-law and be a wonderful grandmother. Finally, I would like to thank my parents Matilde and Tonin Mjeda for their boundless love and for teaching me the value of hard work and persistence.

Concluding, I gratefully acknowledge the financial support of Science Foundation Ireland grant 10/CE/I1855.
Declaration

I hereby certify that the work submitted in this thesis is my own work. It has not been submitted previously to this or any other institute for this or any other academic award. Where use has been made of the work of other people, it has been acknowledged and referenced.

Signed: __________________________

Anila Mjeda

Date: __________________________
# Table of Contents

1. Introduction .......................................................................................................................... 20
   1.1. Motivation ......................................................................................................................... 20
2. Testing of Embedded Automotive Software ........................................................................ 28
   2.1. General Considerations on Testing .............................................................................. 28
      2.1.1. What is Testing? ......................................................................................................... 28
      2.1.2. Coverage ................................................................................................................. 31
   2.2. Embedded Automotive Software .................................................................................. 38
      2.2.1. Automotive Safety-Related Software ...................................................................... 38
      2.2.2. Automotive Embedded Software Testing .............................................................. 42
      2.2.3. Model-Based Testing in the Automotive Industry .................................................. 45
   2.3. A Review of a Selection of Testing Toolsets and Testing Techniques ......................... 50
      2.3.1. AGEDIS .................................................................................................................. 51
      2.3.2. ASML Test Tool ...................................................................................................... 53
      2.3.3. AutoFOCUS ........................................................................................................... 54
      2.3.4. Conformance Kit ..................................................................................................... 55
      2.3.5. CTM and CTM$_{EAB}$ ............................................................................................ 56
      2.3.6. DOORS .................................................................................................................. 57
      2.3.7. GATel .................................................................................................................... 58
      2.3.8. LUTE ...................................................................................................................... 59
      2.3.9. Lustre ...................................................................................................................... 60
      2.3.10. Lutess .................................................................................................................... 61
      2.3.11. MATLAB/Simulink/Stateflow .............................................................................. 62
      2.3.12. PHACT .................................................................................................................. 63
      2.3.13. SCADE .................................................................................................................. 64
      2.3.14. STG ....................................................................................................................... 65
      2.3.15. TestComposer ....................................................................................................... 66
      2.3.16. TGV ....................................................................................................................... 67
      2.3.17. TorX ....................................................................................................................... 68
      2.3.18. TTCN-3 .................................................................................................................. 69
      2.3.19. TVEDA .................................................................................................................. 71
   2.4. Reflections on the Reviewed Tools ................................................................................ 73
   2.5. Conclusions ..................................................................................................................... 75
3. Safety Standards and Formalized Development .................................................................. 78
   3.1. Standards .......................................................................................................................... 78
      3.1.1. IEC 61508 ................................................................................................................. 79
      3.1.2. DO-178B and DO-178C ......................................................................................... 80
      3.1.3. MISRA – Development Guidelines for Vehicle Based Software ......................... 82
      3.1.4. ISO 26262 ................................................................................................................. 82
      3.1.5. AUTOSAR .............................................................................................................. 85
   3.2. Formal Testing ................................................................................................................. 89
      3.2.1. Synopsis of Approaches for Testing FSMs ............................................................ 93
APPENDIX A – Partial Z Specification for an Adaptive Cruise Control .......... 234
APPENDIX B – Illustration of CTM\textsubscript{CONTROL} within the Formalization of a Reactive Testing Process ........................................................................................................... 248
APPENDIX C – Screenshots from the Implementation ....................................... 260
APPENDIX D – MATLAB Code............................................................................. 272
# Table of Figures

Figure 1: Typical ECU-network in a mid-range car (Heiming and Haupt, 2005) .. 40
Figure 2: Product life cycle of a vehicle (Schäuffele and Zurawka 2005) .............. 41
Figure 3: The core process for the development of electronic systems and software, visualised as a V-model (Schäuffele and Zurawka 2005) ....................................... 43
Figure 4: Example of the V-Model in the automotive industry (www.dspaceinc.com) ............................................................................................. 44
Figure 5: Model Based Testing Process (modified from (Lamberg et al. 2004)) ... 47
Figure 6: Taxonomy of Model-Based Testing (Utting et al. 2006, Hamon and Rushby 2007). ......................................................................................................... 48
Figure 7: The type of formality or notation used in tools ........................................ 73
Figure 8: Formality used in industry and academia .............................................. 74
Figure 9: Academic v. industrial use ................................................................. 74
Figure 10: Test coverage delivered by different tools ........................................... 75
Figure 11: Safety Integrity Levels ...................................................................... 79
Figure 12: Key concepts of ISO 26262 ................................................................ 83
Figure 13: Recommended methods for verification of requirements by ISO 26262 (Part 6) ................................................................................................. 84
Figure 14: Structural coverage metrics recommended by ISO 26262 (Part 6) ...... 85
Figure 15: AUTOSAR’s Layered Software Architecture (AUTOSAR development partnership 2013) ......................................................................................... 86
Figure 16: System $S$ .......................................................................................... 109
Figure 17: Test cases generated for the control aspect of $\text{CTM}_{\text{CONTROL}}$ .......... 111
Figure 18: $\text{CTM}_{\text{CONTROL}}$ mapped ................................................................. 112
Figure 19: The Taxonomy of Reactive Testing .................................................... 116
Figure 20: The abstract and concrete aspects of illustrating $\text{CTM}_{\text{CONTROL}}$ .......... 123
Figure 21: Workflow of the proposed usability model ......................................... 123
Figure 22: Proposed model based testing taxonomy ........................................... 138
Figure 23: Overview of the Controlled Experiment Model .................................... 149
Figure 24: Overview of the state machine ......................................................... 150
Figure 25: Tests signals, designed to isolate the transition of interest ................. 151
Figure 26: Tests generated via classic CTM ....................................................... 152
Figure 27: The MC/DC aspect of the classification tree of $\text{CTM}_{\text{CONTROL}}$ .......... 153
Figure 28: The combinations algorithm ............................................................. 154
Figure 29: Test signals generated via the CTM approach .................................... 154
Figure 30: Test signals generated via the MC/DC aspect of the $\text{CTM}_{\text{CONTROL}}$ approach ................................................................................................................ 155
Figure 31: Coverage achieved when using the classic CTM method .................. 156
Figure 32: Complete coverage of the logical statements inside the transition ...... 157
Figure 33: A roadmap for the ACC case study .................................................... 159
Figure 34: Schematic overview of an ACC (FORD 2012) ................................... 160
Figure 35: ACC interface ................................................................................... 160
Figure 36: Conceptual view of the ACC prototype ............................................. 161
Figure 37: The SUT Statechart ........................................................................... 163
Figure 38: Notation used inside the SUT’s states ................................................. 164
Figure 39: Notation used in the SUT’s transitions (MathWorks 2003, Pretschner et al. 2005, Conrad and Fey 2005) ............................................................... 164
Figure 40: Implementation of the funcCalcVel() as a graphical function .......... 166
Figure 41: Rationale behind brake by distance .................................................... 167
Figure 42: Implementation of funCalcDistance() as a graphical function .......... 168
Figure 43: Data Partition Pattern Model .............................................................. 169
Figure 44: Data Partition Pattern ...................................................................... 169
Figure 45: The Partition Pattern ........................................................................ 170
Figure 46: Graphical Implementation of the Classification Tree ....................... 171
Figure 47: Scope showing the input signal filtered in three partitions ............... 172
Figure 48: A view of the test harness for the ACC ............................................ 173
Figure 49: Conceptual view of the components of the Test Generator ............... 174
Figure 50: The implementation of the reactive test abstract state .................... 175
Figure 51: Inside the Test Mode Control Layer ............................................... 175
Figure 52: Conceptual View of the Test Signal Creation Layer ......................... 176
Figure 53: Case study test reactivity mapped into the test reactivity taxonomy ... 178
Figure 54: Example of MATLAB syntax to read data from an Excel file .......... 183
Figure 55: GUI for the ACC test harness ............................................................ 184
Figure 56: Generation of the TestDataMatrix ..................................................... 184
Figure 57: CTM classic ...................................................................................... 187
Figure 58: The MC/DC only coverage with all permutations ......................... 187
Figure 59: The full CTMCONTROL coverage ............................................... 188
Figure 60: Conceptual view of the structural logic of the analysis ..................... 191
Figure 61: CTMCONTROL mapped ................................................................. 193
Figure 62: CTMCONTROL mapped into the MBT taxonomy ......................... 199
Figure 63: Recommended methods for verification of requirements by ISO 26262 Part6 ................................................................. 202
Figure 64: Structural coverage metrics recommended by ISO 26262 Part6 ....... 203
Figure 65: Some of the competing forces, this research focused into ............... 209
Figure 66: The CTMCONTROL and FRTP connection ................................. 213
Figure 67: A snapshot of the synergy among the contributions of this research .. 214
Figure 68: Case study test reactivity mapped into the test reactivity taxonomy ... 250
List of Tables

Table 1: A Snapshot of Research Contributions .......................................................... 26
Table 2: Software testing in the context of different testing hypotheses ............... 30
Table 3: The reviewed toolsets .............................................................................. 51
Table 4: AGEDIS .................................................................................................. 53
Table 5: AsmL Test Tool ...................................................................................... 54
Table 6: AutoFOCUS ........................................................................................... 55
Table 7: Conformance Kit .................................................................................... 56
Table 8: CTM ...................................................................................................... 57
Table 9: DOORS .................................................................................................. 58
Table 10: GATel .................................................................................................. 59
Table 11: LTG ...................................................................................................... 59
Table 12: Lurette .................................................................................................. 60
Table 13: Lustre .................................................................................................... 61
Table 14: Lutess ................................................................................................... 62
Table 15: The M/S/S Environment ...................................................................... 63
Table 16: PHACT ................................................................................................. 64
Table 17: SCADE ................................................................................................ 65
Table 18: STG ....................................................................................................... 66
Table 19: TestComposer ....................................................................................... 66
Table 20: TGV ...................................................................................................... 67
Table 21: TorX ....................................................................................................... 68
Table 22: TTCN-3 ............................................................................................... 71
Table 23: TVEDA ................................................................................................ 72
Table 24: Tests generated by the control aspect of CTM ................................. 111
Table 25: Weighted partitions in tabular form ................................................... 136
Table 26: I/O Parameters of the SUT .................................................................. 162
Table 27: SUT Transition Labels ....................................................................... 165
Table 28: InitSetSpeed (Vehicle Velocity) ......................................................... 181
Table 29: Non-fully parameterized set of values .............................................. 182
Table 30: The parameter values considered for the MC/DC .............................. 182
Table 31: Test data used for both test approaches ........................................... 183
Table 32: MC/DC coverage compared ............................................................... 196
Table 33: InitSetSpeed (Vehicle Velocity) .......................................................... 253
Table 34: TargetVehicleVel (Velocity of the vehicle ahead) .............................. 254
Table 35: TargetPresent (Vehicle Ahead detected flag) ..................................... 255
Table 36: LeverPos (ACC lever position) ............................................................ 256
Table 37: Pedal flag (Pedal pressed flag) ............................................................ 257
Table 38: Non-fully parameterized set of values .............................................. 258
Table 39: The parameter values considered for the MC/DC .............................. 258
Table 40: Test data used for both test approaches ........................................... 259
PART 1
Introduction and Literature Review
1. Introduction

“It was one of those cases where you approve the broad, general principle of an idea but can't help being in a bit of a twitter at the prospect of putting it into practical effect. I explained this to Jeeves, and he said much the same thing had bothered Hamlet.”


This chapter discusses the motivation and research questions that underpin the research reported in my PhD thesis. Following that, the scope and contributions of the research are presented. The chapter concludes by delineating the structure of the thesis.

1.1. Motivation

Society today relies on “smart” products which are expected to seamlessly facilitate a growing array of tasks. This, admittedly contestable, simplification of everyday life is frequently delivered by evermore complex software. Software is routinely used in safety-critical applications in medical devices, control of nuclear plants, avionics and automotive applications to name but a few. This class of software must conform to the state of art in terms of scientific and technological maturity, so prevention of product liability claims requires, at minimum, adherence to the relevant quality assurance (QA) standards.

The area of interest of this thesis consists of model-based testing for safety-related software with a particular focus in automotive software.

In-vehicle embedded software is increasingly responsible for delivering added functionality and in the process is becoming more complex. In this domain, software defects can have quite drastic consequences. Automotive software is often developed in the absence of the actual hardware where it will be deployed and its development process needs to facilitate interdisciplinary communication and understanding. Clear and rigorous
notations and practices are needed to facilitate cross-disciplinary communication among hardware engineers, system engineers, software developers, software testers, vehicle test drivers, etc.

Automotive software’s input space is made of discrete events and a mixture of continuous and discrete signals and data. Testing approaches and tools need to be able to deal with this hybrid universe of discourse.

There can be a considerable timespan of around 15 years (Schäuffele and Zurawka 2005) from when automotive software is designed, to when the vehicle is taken off the road. Furthermore, automotive software is routinely deployed in different vehicle versions in a business and technological setup that does not allow for easy updates or ‘patches’ after release.

In the mid-nineties, a great number of automotive companies went through a drive of specialising quite narrowly and hence relying on outsourced expertise for everything outside their field of specialisation. In other words most if not all automotive software needs to go through some third party acceptance tests and certification before passing through the supply chain. Furthermore, as is often the case with complex systems, automotive software requires a great number of tests, long running test sequences and tests that need to be easily repeated at different stages of the product development. The above factors, contribute to the need for test automation.

Typically, automotive software is installed into Electronic Control Units (ECUs) and needs to communicate via networks with an array of sensors and actuators. As a result, testing activities need to cater for issues of software-hardware interaction such as time lag and propagation of hardware faults. Testing is especially difficult because embedded automotive software needs to operate within time-sensitive functionality constraints.

---

1 The terms tools and toolsets are used interchangeably throughout this thesis.
2 The test activities need to be able to cater for hard timing requirements such as “a precise sequence of time-sensitive actions needs to happen within a time interval of 20 μsec”.

In practice, the inherent complexity of automotive embedded software is typically handled via building functional models based on the functional specification for the system (Stürmer et al. 2007); instead of handwriting code, developers of embedded automotive software use graphical modelling and simulation in integrated environments such as MATLAB/Simulink/Stateflow (M/S/S) (Lehmann and Krämer 2008). Reportedly, around 50% of functional behaviour for embedded systems is modelled in the M/S/S environment (Helmerich et al. 2005).

To date, there is considerable research reporting on the benefits of model-based development and its well-established position in the automotive industry (Broy et al. 2007, Conrad and Fey 2005, Pretschner et al. 2005). Yet, despite the affinity of model-based development with the automotive industry, model-based testing tends to be done in a bespoke and non-systematic fashion using a number of difficult-to-integrate proprietary systems with only a few processes that are specifically tailored to model-based testing of automotive software (Lamberg et al. 2004, Lehmann and Krämer 2008, Zander-Nowicka 2009).

Current moves towards standardization such as AUTOSAR (AUTOSAR 2006) and the QA requirements of the standards that regulate automotive software such as IEC 61508 (IEC 1998) and ISO 26262 (ISO 2011) increase the need for reliable and confidence-inspiring testing approaches. These testing approaches need to be specifically targeted at the embedded automotive software sector. Additionally, in order to address the increased QA demands for safety-critical software, the standards (IEC 1998, ISO 2011) require the developers to make use of the rigour and the robust capabilities of formal methods and processes.
1.2. Research Questions

This research work was motivated by two main issues:

1. There is a need for improved testing methods for embedded safety-critical software. There is a considerable amount of academic research work in this area (Broy 2003, Dai 2004, Conrad and Fey 2005, Pretschner et al. 2005, Broy et al. 2007, Zander-Nowicka 2009, Utting et al. 2010). However, the unfamiliarity of practitioners with tools proposed by researchers, built in an academic setup and catering for idealised conditions, creates a real or perceived barrier for uptake by industry (Dill and Rushby 1996, Holloway and Butler 1996, Broy 2009).

2. QA standards regulating the field urge practitioners to harness the benefits of formal notation, methods and processes (ISO 2011, IEC 1998). Yet, formality is somewhat perceived to be unworkable or to be an insurmountable barrier for the typical tester (Dill and Rushby 1996, Holloway and Butler 1996, Broy 2009, Bowen and Hinchey 2006).

In pursuit of trying to find solutions for the above issues, I undertook an in-depth analysis of the field’s state of the art. As a result, the research questions are:

1. Can a software testing method, which is in common use in industry be improved in such a way that brings that method closer to the latest QA requirements?
2. Can this proposed method be implemented via toolsets already popular in the automotive industry?
3. Can the improved method be used within a software testing process, which incorporates formality?
4. Finally, how can the worthiness of the proposed method be evaluated?

In order to better delineate the scope of this research, the next section presents the evaluation criteria, which are used to evaluate the contributions reported in this thesis.
1.3. Evaluation Criteria

One of the challenges of this research is the fact that it spans over a relatively large number of disciplines.

The disciplines incorporated in this research include QA standards regulating automotive software; testing of embedded software in general and of embedded automotive safety-critical software in particular; formal testing and notations; mathematical modelling and simulation; signal theory and signal properties; control theory; modelling of state-based behaviour systems via graphical approaches; and, design, programming and simulation in MATLAB, Simulink and Stateflow.

From the above, the first three disciplines form the relevant areas in literature, which are within the scope of this research. A review of these can be found in the Chapters 2 and 3 of this thesis.

The other disciplines, namely formal notations, mathematical modelling and simulation, signal theory and signal properties, control theory, modelling of state-based systems and design, programming and simulation in M/S/S form part of the skills, means and toolsets needed to achieve the aim of this research.

Based on the gaps identified in existing approaches, the challenges currently faced by the industry, and the focus of this research, the following evaluation criteria have been identified for the contributions reported in this thesis:

1. The proposed testing method should target a specific requirement or a number of requirements posed by QA standards (ISO 2011, IEC 1998).
2. The improvement of the proposed method over an existing method, should be measurable, in order to facilitate quantitative measurement in an experimental setup.
3. In order not to introduce a steep learning curve, which could serve as a barrier for its uptake by industry, the method should make possible the
use of toolsets that are already used in industry (deduced from research reported by (Hinchey 2003, Broy 2009, Parnas 2010)).

4. The method should incorporate graphical modelling for software in order to facilitate communication and understanding as early as possible in the development process. This is seen as especially important in the context of the multidisciplinary nature of automotive software development, which typically incorporates aspects from mechanical, electrical and software engineering disciplines (Broy 2009).

5. The method should allow the benefits of formal notation, methods and processes to be harnessed (to cater for safety-critical systems or safety-critical functions in a system). The incorporation of formality should be flexible and introduced as needed. It should be ‘tailorable’ to the needs of the company or the project at hand (Dill and Rushby 1996, Holloway and Butler 1996, Broy 2009, ISO 2011, IEC 1998).

1.4. Research Contributions

The contributions of this research are presented in detail in Chapter 4, validated in Chapter 5, and analysed in Chapter 6 of this thesis. For ease of reference, the contributions are synthesised in the table below (Table 1):

Table 1: A Snapshot of Research Contributions

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A new testing method:</strong></td>
<td>This research proposes a testing method called CTM\textsubscript{CONTROL}, which enhances the classic Classification Tree Method, by incorporating Modified Condition / Decision Coverage (MC/DC) capability. The proposed method targets MC/DC coverage as one of the objectives posed by the relevant QA standards. CTM\textsubscript{CONTROL} is implemented in the M/S/S environment, which is commonly used in industry.</td>
</tr>
<tr>
<td><strong>The Formalization of a Reactive Testing Process.</strong></td>
<td>The thesis provides the steps of the Formalization of a Reactive Testing Process (FRTP) that can be tailored to the specific needs of a project or company. The thesis also illustrates how CTM\textsubscript{CONTROL} can be used within this formalized process.</td>
</tr>
<tr>
<td><strong>Enhanced taxonomy for model-based testing and a new taxonomy of reactive testing:</strong></td>
<td>Within the FRTP, the thesis proposes enhancements on a previously published model-based testing taxonomy (Utting et al. 2010, Zander-Nowicka 2009) and it also proposes a new taxonomy of reactive testing.</td>
</tr>
</tbody>
</table>
1.5. Structure of the Thesis

This thesis is organized in three parts. The first part, in addition to the “Introduction”, is composed of Chapters 2 and 3 and presents a review of the literature for the area of interest.

Chapter 2 named “Testing of Embedded Automotive Software” discusses some general considerations on testing and the state of art for automotive software testing. A number of testing tools and testing techniques are discussed and analysed.

Chapter 3 named “Safety Standards and Formalized Development” discusses a number of relevant quality assurance standards and formal testing approaches. It pays a particular attention to graphical approaches geared towards state-based behaviour.

The second part contains Chapter 4, named “Research Contributions”, which presents the contributions of this thesis.

The third part contains the Chapters 5 to 7 and presents the validation of the proposed method, an evaluation and analysis of the contribution of the thesis, and some conclusions on the achievements and limitations of the research presented.
2. Testing of Embedded Automotive Software

“No amount of experimentation can ever prove me right; a single experiment can prove me wrong”.

*Quote attributed to Albert Einstein.*

This chapter discusses some of the concepts surrounding the field of testing in general. It includes a discussion on automotive safety-related software, and, model-based testing of embedded automotive software. The chapter also reviews and analyses twenty testing toolsets and testing techniques, which incorporate different degrees of formality.

2.1. General Considerations on Testing

2.1.1. What is Testing?

Among the software engineering theoreticians, testing as an activity finds an array of definitions. In the course of this thesis, when in need of a generic definition for testing, it will be SWEBOK’s (IEEE Computer Society 2005) one: “*Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems*.”

While there is a consensus over the idea that testing cannot be used to show the absence of errors famously from (Dijkstra 1974) and an array of other

---

3 Guide to the Software Engineering Body of Knowledge - SWEBOK
4 In this document, the term *error* is used to refer to the cause of a specific malfunction. The same notion is often found in literature being referred to by several names including *bug, fault or defect*. That said, in some schools of thought a *bug* or a *defect* is what results from an *error*, while the terms *fault or failure* are used to describe the evidence of a bug’s existence. Since this granularity of terms aimed at discerning between causes, effects and the symptoms targeted by testing does not interfere with the ideas discussed and proposed in this thesis, the terms *error, bug, fault or defect* will be used interchangeably to convey the same meaning.
authors, which includes (Kaner et al. 1999, IEEE Computer Society 2005, Sommerville 2007), the goal of testing finds different definitions. Pretschner (2005) argues that “Testing comprises activities that aim at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: observable differences between the behaviours of implementation and specification”. Koopman (2007) refers to the idea of testing different hypotheses, which include “Testing the hypothesis that there are no major bugs”. Sommerville, argues that “Software testing has two goals: - To demonstrate to the developer and the customer that the software meets its requirements; - To discover faults or defects in the software where the behaviour of the software is incorrect, undesirable or does not conform to its specification.”

In other words, Sommerville includes testing activities that do not necessarily aim to uncover defects, which are used in the context of validation testing (checking the system against its requirements). In this regard, this research will take the view that testing is an activity that aims to evaluate two hypotheses:

1. There are errors
2. There are no major errors

While a framework that defines what exactly makes an error a major one is outside the scope of this thesis, it could be deduced that this classification is influenced by the safety-critical requirements of the system.

In this thesis, the success of a testing activity will be judged according to the following truth table:
Table 2: Software testing in the context of different testing hypotheses

<table>
<thead>
<tr>
<th>Testing Hypothesis</th>
<th>Bugs Found</th>
<th>Software Testing Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are errors</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>There are no major errors</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Here, we are abiding by the Aristotelian two-valued logic. Any undefined logic values that these testing hypotheses may lead to, will be considered to evaluate to False.

The inclusion of testing activities that aim to evaluate the hypothesis that they are no major errors does not contradict Dijkstra’s (1974) famous aphorism that “program testing can be used to show the presence of bugs, but never to show their absence”. This it is meant to acknowledge the existence of these testing activities typically performed at the very last stages before product release, and, in these occasions, to relate a positive result to a testing activity that has not found any major errors.

Software development companies commonly are under pressure to maximize the operational capabilities of the software developers (software testers included) so as to be able to produce more, while increasing quality and preferably reducing overall costs. The concepts of effectiveness and efficiency are often seen as important indicators to measure the ‘business health’ of the testing process in its entirety. Test efficiency is commonly measured by dividing the number of defects found in a test by the effort needed to perform the test (Pfleeger 2001). The effort here usually incorporates the element of time (Kan 2002). Whereas test effectiveness is often defined as the number of defects found through software testing divided by the total number of defects and it is related to the extent of the testing impact (Kan 2002). Here it is important to highlight that test effectiveness can only be understood in the context of the objectives of a specific testing process (IEEE Computer Society 2005).
The concepts of test efficiency and effectiveness are briefly mentioned here for completeness but are not in the focus of this thesis. Coverage is another interesting concept, which is quite central to software testing. Coverage is also closely related to this research and it is discussed in the following section.

2.1.2. Coverage

One of the most challenging features inherent to testing is the unfeasibility and very often the impossibility of complete testing for software. This creates the need to quantify the completeness of testing in a manner that can inform the decision on when to stop the testing process. Coverage, understood as the notion of how much of the software product has been covered by testing, is one measure that comes up frequently when discussing completeness of testing (Kaner and Bach 2005).

While it is possible to find commercial products that are advertised to offer “100 % test coverage”, the reality is that this claim usually represents only a small fraction of what can be covered by testing. Usually, it can refer to one dimension of coverage, such as all the programming branches were tested, this, while ‘covering’ all branches of programming does not translate into 100% test coverage and even less to complete testing. For example, covering all of the branches in a program does not necessarily test whether the product conforms to its specification.

Coverage is a multidimensional concept and to highlight this fact, in the nineties, Kaner (1996), presented a non-exhaustive list of over a hundred dimensions of what one can cover while testing a software product. Conformance to specification, correctness, usability, boundary conditions, performance, state-transitions, mainstream usage (against scenarios), load testing (events, memory usage, error rates), error recovery, security, compatibility/configuration (equipment variations; old versions), installability/serviceability, are just a few possible types of coverage listed by Kaner (1996).
Every one single aspect of coverage is inadequate to capture the adequacy of testing and overall, achieving a good and the right kind of coverage underpins a good testing strategy but is not sufficient to provide good testing on its own (Koopman 2007).

Among all the dimensions of coverage, structural coverage is of particular interest for this research. Here it is important to highlight that the concept of structural coverage can be used in very different capacities. For example, it is commonly used in structural testing, where it is understood as a set of activities that exercise the software with tests generated based on the source code. Whereas, it is of particular interest for this research its use in the context of structural coverage analysis, which is a measure to “determine which code structure was not exercised by the requirements-based test procedures” (RTCA 2011). In other words while structural testing is concerned with the source code, structural coverage analysis is concerned with the requirements of a software product.

The concept of structural coverage analysis is used to judge the compliance of a specific testing process to the respective regulating QA standards for safety-related software such as DO-178B (RTCA 1992) and ISO26262 (ISO 2011). Within the context of structural coverage analysis, of particular interest for this thesis is the Modified Condition / Decision Coverage (MC/DC) criterion, since it is a criterion required by the ISO 26262 ISO26262 (ISO 2011) standard that regulates automotive software. The MC/DC criterion is discussed in the following section.

2.1.2.1. MC/DC

MC/DC was initially proposed in (Chilenski and Miller 1994) and it focuses on faults that may be introduced in logical expressions. The first standard to include MC/DC in its requirements was DO-178B (RTCA 1992). DO-178B (RTCA 1992) specifies that to meet the MC/DC criterion: “Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at
least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible conditions.”

According to DO-178B’s definition a condition is a Boolean expression containing no Boolean operators. Whereas a decision is a Boolean expression that is composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is considered to be a distinct condition.

The MC/DC criterion defined as above is commonly known as the unique-cause MC/DC. Crucially, the unique-cause approach cannot be achieved for strongly coupled conditions. Coupled conditions, in this context, are conditions that are interdependent. The conditions are strongly-coupled when changing the truth value to one of the conditions always changes the truth value of the other condition. They are weakly-coupled when changing the truth value of one condition sometimes changes the truth value of the other condition.

For example for the logical structure: (A AND B) OR (A AND C), the unique-cause MC/DC approach would require to toggle the value of the first A between true and false, while holding the second A constant. This presents a mathematical impossibility.

The technique of masking MC/DC helps to mitigate part of this issue. In this context, masking refers to the concept that specific combinations of logical operators can mask (hide) the effect of some of the inputs to the logical construct. For example, a false input to an AND operator masks all other inputs. Similarly a true input to an OR operator masks all other inputs. The masking approach to MC/DC, which is commonly called masking MC/DC, allows to change more than one of the inputs of the coupled conditions at a time, as long as it can be observed that the condition of interest is the only condition that affects the value of the outcome of the decision.
For example in masking MC/DC, to show the independent effect of C for the logical construct (A OR B) AND (C OR D), the logical construct (A OR B) must be *true* since otherwise the value of the decision's outcome will always be *false*. For unique-cause MC/DC, the values of A and B must be kept fixed when showing the independent effect of C. While masking MC/DC allows A and B to change values, as long as the outcome of (A OR B) is *true*. In this way, the masking approach allows to generate more executable tests that meet the MC/DC criteria.

Masking MC/DC is considered as an acceptable method for meeting MC/DC according to the objectives of DO-178B (CAST 2001).

Vilkomir and Bowen (2006) have provided a Z formalization for MC/DC which considers both the unique-cause MC/DC and the masking MC/DC. Vilkomir and Bowen have also addressed one of the inherent limitations of MC/DC: it lacks requirements to analyse the possibility of any unwanted activations of the decisions (operation without demand) caused by some erroneous ‘hidden’ logic in the SUT. They cater for this shortcoming by providing a more stringent criterion named Reinforced Condition / Decision Coverage (RC/DC). (Vilkomir and Bowen 2006), define RC/DC as: “*Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, each condition in a decision has been shown to independently affect the decision’s outcome, and each condition in a decision has been shown to independently keep the decision’s outcome. A condition is shown to independently affect and keep a decision’s outcome by varying just that condition while holding fixed (if it is possible) all other conditions*”.

Kirner (2009) has also provided a formalization for MC/DC, which is the equivalent of (Vilkomir and Bowen 2006) with the only difference that Kirner also presents some formal criteria to support MC/DC preservation for code transformations. The code transformations considered are those that change the reachability of statements and conditions; and, the transformations that add new conditional control-flow paths into the
program. Whereas, Cormar et al (2010) use Binary Decision Diagrams (BDDs) to characterize a number of Object Branch Coverage (OBC) techniques that can be used to claim MC/DC coverage.

The focus of Kirner and Cormar et al. is in respectively code-level and object-level criteria for MC/DC. Whereas, the formalization presented in (Vilkomir and Bowen 2006) is equally concerned with specification-based and code-level MC/DC. Hence this thesis will refer to the definition provided by (Vilkomir and Bowen 2006) as the formal definition of MC/DC.

ISO26262 (ISO 2011), which is the standard that regulates automotive software, requires MC/DC testing for safety-related software but it does not formally define MC/DC. Despite the formalization efforts mentioned above, ISO26262 has not adopted any particular formalized definition of the criterion, leaving its implementation somewhat open to interpretation.

Every coverage metric (code or specification based) is affected by the visibility of the testing process or tester to the specific aspect of the software product, which is subject to that metric. Visibility in software testing is often described in terms of colours. Depending on the opacity of the code to the tester, testing is often described by a specific grey-scale hue in an (artificial) white to black spectrum. A white box metaphor conveys the idea that the box is transparent and every single detail of the code is visible and, the darker the box, the less one can see from the code. These different colours of testing are discussed in the next section.

2.1.2.2. The Colours of Testing:

In white-box testing, which is also known as structural testing, the implementation and the internal structure or the system is known to the tester. Coverage in white-box testing takes dimensions such as statement coverage, branch coverage, condition-decision coverage, etc.

According to IEEE Standard Glossary for Software Engineering Technology (IEEE 1990) :

"Black-box testing (also called functional testing) is testing that ignores the internal mechanism of a system or
Keeping with the metaphor of the box for the system under test, the tester will typically know what the system does, what goes in, what comes out of the box, how does the system ‘talk’ at the interface level while not being able to see inside the box (with no knowledge on how the system has been implemented). In this type of testing the structure of the program is not considered.

Black-box testing is also called behavioural testing since it typically relies on the functional requirements of the software and involves the observation of the output and more generally the overall behaviour of the system under test. It aims to exercise all of the relevant functional behaviour of the software. Black-box testing targets errors such as incorrect or missing functionality, interface errors, errors in data structures used by interfaces, behavior or performance errors, initialization and termination errors (Pressman 1996).

All the reasoning is based on the functional requirements for the system. The tester’s problem in black-box testing is to select the right tests, which are more likely to reveal the defects of the functional behaviour of the software. Typically the number of possible tests is too large hence the key problem for the tester, again, is which tests to select. Coverage in black-box testing, takes the dimensions of the percentage of requirements tested, the parameters that are tested, etc.

Obvious advantages like the lack of need to disclose the implementation of a system; the ability to design tests independently from the system’s implementation; as well as the ability of testing different implementations of the same system with minimized changes, and outsourcing trends make black-box testing a very good match for the automotive industry. In the automotive field, often a tester would be interested to know for example if a specific ECU received by a supplier functions according to the requested
specification; without having any knowledge on how exactly that ECU’s functionality was implemented. In fact in the automotive industry it is quite appealing for Original Equipment Manufacturers (OEMs) and their suppliers to communicate via black-box models (Ahluwalia et al. 2005, Pretschner et al. 2005, Pretschner et al. 2007). Especially, since these models can be used to generate tests (Pretschner et al. 2005), and to judge if the system conforms to its specification.

In many occasions it is difficult to categorise an industrial testing method as strictly black or white box. For example, there might be cases when the tester knows a certain level of detail on the structure of the system under test even though it does not know the implementation code. This type of testing is usually called grey-box testing.
2.2. Embedded Automotive Software

2.2.1. Automotive Safety-Related Software
Software delivers an increasing variety of functionality in a number of safety-related systems, which include medical devices, avionics and automotive systems. In other words, in systems where software malfunctions could cause or contribute to fatal accidents. Software that delivers part of such safety sensitive functionality, typically, has to be developed in accordance with the latest set of QA standards or guidelines required by the respective industry. The current QA standards can vary quite considerably between different fields and industrial sectors and an analysis of the standards, which govern or affect embedded automotive software can be found in Section 3.1 of this thesis.

Embedded systems are a key factor in the automotive domain where the European industry is often identified as the market leader (Helmerich et al. 2005). The embedded automotive software, which is in the focus of this research, caters for an impressive range of features in the today’s vehicle. In fact software has become the driving force behind innovation and makes up an estimated 40%-50% of value creation in automotive electronics (Pretschner et al. 2007, Hansen 2004, Helmerich et al. 2005).

The automotive industry has had its fair share of prominent product faults and recalls. In the last few years, incidents such as engines that stalled unexpectedly in fast flowing traffic or ‘stuck’ acceleration pedals, gained world-wide notoriety. For example, in 2004, DaimlerChrysler had to recall around 700 000 cars with Sensotronic Brake Control (SBS), which in some instances could unexpectedly shut down and affect the braking distance. The recall also resulted in legal proceedings and involved Bosch that had manufactured the hardware for the SBS. In 2010, Toyota had two different very public concerns with its cars. The first involved unintended acceleration in cars and the second a fault in Engine Control Modules (ECM), which reportedly could cause the engine to stall while the car was being driven. The unintended acceleration was claimed by Toyota to be due to a moving floor mat, while other parties suggested that underlying
electronic faults were also involved. The faulty ECM-s caused a voluntary recall by Toyota on a number of Toyota Corolla and Corolla Matrix cars.

The reality is that automotive embedded software is a very demanding and complex area that expects software to operate in harsh physical conditions that include high temperatures, noise, vibrations and the possibility of electromagnetic interference. Embedded automotive software typically needs to interface with a number of distributed sensors and actuators, cater for hybrid behaviour and deliver critical hard time functionality.

The automotive industry uses a unit cost model and operates in a highly competitive market where keeping a low unit cost is one of its biggest pressures. The engineers are typically under time-to-market pressure trying to deliver more functionality from a competitively priced unit.

In terms of software solutions, this translates into deploying the software in electronic units with a restricted memory and computing capacity. First of all this means that there can be no single electronic unit that handles all of the functionality. The automotive industry, has tackled this by using several low-cost and low-memory electronic units that communicate via networks. The automotive software resides into several ECUs, which communicate via in-vehicle networks according to predefined communication protocols. The ECUs are widely distributed through the car\(^5\) as illustrated in Figure 1, which shows the ECU network in a mid-range Ford Focus (Heiming and Haupt 2005).

---

\(^5\) In addition to other requirements, in some cases the ECUs have to be placed in proximity of the actual hardware that is governed or influenced by the functionality they deliver.
The first network protocol used in cars was CAN (Controller Area Network) (ISO 2003), which is a two-wire bus that uses asynchronous transmission with information passed from transmitters to receivers via data frames controlled by start and stop bits at the beginning and end of each character. Today’s high end cars can have over 70 ECUs and one of the challenges of this distributed architecture is the timely communication among them. One of the manners in which the automotive industry has at some level addressed the issue of message latency has been the development of time synchronous in-vehicle bus protocols such as TTCAN (ISO 2004), and FlexRay (Paret 2012).

The depicted ECUs are: (ABS: Antilock Brake System (ABS/ESP); ACU: Audio Control Unit (incl. LowLevelNAV); CDDJ: CD Changer; DDCU: Driver Door Control Unit; DISP: Display; EATC: Electronic Automatic Temperature Control; EHPAS: El. Hydraulic Power Assist Steering; EPB: Electric Park Brake; FACM: Fuel Additive Control Module; FFH: Fuel Fired Heater; HCM: Headlamp Control Module; HEC: Hybrid Electronic Cluster; KVM: Keyless Vehicle Module; NAV: Navigation Module; PAM: Parking Aid Module; PCM: Powertrain Control Module; PDCU: Passenger Door Control Unit; PJB: Passenger Junction Box (GEM); RCM: Restraints Control Module; RLDCU: Rear Left Door Control Unit; RRDCU: Rear Right Door Control Unit; RSE: Rear Seat Entertainment; SAS: Steering Angle Sensor; TCM: Transmission Control Module; VRM: Voice Recognition Module; YAW: Yaw rate/Lat. Acc sensor cluster).

Originally invented by Bosch in 1985.

Other networks commonly found in cars include LIN (Local Interconnect Network) and MOST (Media Oriented Systems Transport). LIN is a comparatively cheap network. It offers slow transfer rates of 10-20 Kbit/s and is typically used for sub-bus applications such as communication between ECUs.
Another feature of automotive software is the co-existence of long product life with short development cycles. Vehicles tend to have a product life cycle of about twenty-five years (Schäuffele and Zurawka 2005), from these typically three years would be spent into the development phase, seven years into the production phase and finally ten to fifteen years into the operation and service phase (Fig. 2).

This can be at odds with the short development cycle of automotive software.

It is obvious that a software program does not degrade from use over time. The situation is very different for hardware, with degradation from use being the norm. Hence especially when delivering critical functionality, the software has to be highly reliable and furthermore needs to abide by fault models that expect the hardware to fail. If possible, in the circumstances of system failures the system needs to fail safely. QA in cars is a very big, expensive and challenging field. In terms of QA expectations, automotive software has to contend with the avionics industry where safety critical software has delivered $10^9$ hours without failures. The automotive QA

---

as for example the communication to and from a car’s electrical mirrors. The MOST bus is a multimedia plastic optical fibre, which is used for multimedia applications and it provides transfer rates of up to 24.5 Mb/s. Both of these networks typically use a gateway interconnection to the vehicle bus (such as CAN).
standards have introduced the same reliability requirements of $10^9$ hours mean time between failures for its most critical software. In reality, for many automotive safety critical systems the respective reliability numbers are not even known (Broy 2006) and further research into measuring and improving reliability is needed.

While testing alone cannot possibly address all the quality aspects of software, it can certainly give a considerable contribution and there is a need for more sophisticated and powerful testing methodologies to support the specific needs of automotive software. Especially in view of the current trends for standardization, such as AUTOSAR, which open the market for new suppliers of automotive software, the need for reliable and confidence inspiring testing methodologies becomes even more pressing.

2.2.2. Automotive Embedded Software Testing

The development process in the automotive field commonly follows what is known as the V-model$^9$. The name is derived from the fact that the flow of the development steps from specification, consideration, to final product completion, can be likened to the shape of the letter “V” (Fig. 3). The left tail of the “V” represents the analysis and specification stream where the system specifications are defined. The right tail of the “V” represents the testing stream where the system is being integrated and tested (against the specifications defined on the left-tail). The bottom of the “V”, where the tails meet, represents the development stream.

---

$^9$ The V-Model was originally developed by Industrieanlagen-Betriebsgesellschaft GmbH (IABG) in Ottobrunn near Munich on behalf of the Federal Ministry of Defence (BMVg) and in cooperation with the Federal Office for Defence Technology and Procurement (BWB) in Koblenz, Germany. In 1992, it was taken over by the Federal Ministry of the Interior for the civilian Federal Administration domain and has been a binding obligatory regulation since June 1996 also for this sector. The documentation and directives regulating the V-Model are available at http://www.v-modell.iabg.de/.
In this process, a developer must repeatedly verify the presence or absence of faults in the system, make the needed modifications if any are found and then verify the presence or absence of faults in the updated version of the system. A good example of the V-model used in the automotive software realm is the ECU development process as outlined by D-Space¹⁰ (Fig. 4).

¹⁰<http://www.dspaceinc.com>
Figure 4: Example of the V-Model in the automotive industry (www.dspaceinc.com)

The coarse view of the model might give the impression that the testing activities first start only after the control design activities but activities in the opposing sides of the “V” actually serve as a base for one another. What is more, each of the steps is typically performed several times. So, during the ‘Control Design’ phase the developer can also design functions and perform initial validation procedures. Similarly during ‘Function Prototyping’ several tests can be run off-line or in an actual vehicle or test bed. During the ‘Automatic Production Code Generation’, the prototyped functions are converted into code and deployed in the ECU. Testing and calibration tasks can be run through each one of the other phases.

In the automotive industry, it is common to test how a specific module of a system will function when possibly a number of the other components of the system (both hardware and software) have not been built yet. This contributes to the fact that the automotive industry’s software development process relies heavily in simulation and it typically revolves around X-in-the-Loop practices (Pretschner et al. 2007). Where the term X-in-the-Loop covers:
– Hardware-in-the-Loop (HiL), where testing involves interfacing with the intended hardware controller while running the tests.
– Model-in-the-Loop (MiL) testing approach, where there are no hardware components, but the system under test and the environment with which it will interact are simulated as models.
– Software-in-the-Loop (SiL) testing practices, which involve some portion of software being tested in a simulated environment, typically without any hardware components present.
– Processor-in-the-Loop (PiL) testing, where the software runs in target boards which have the target processor or within a simulated target board and processor.

Most of the car manufacturers have the necessity to present to the market large numbers of variants and configurations for each vehicle model. This requires, among other things, good reusable testing models that are especially good in checking for unintentional feature interaction or unwanted activations introduced as a result of the integration of new elements. In the very cost aware automotive industry, the systems are often simulated very early in the development process (Broy 2006) and model-based development, which includes model-based testing, is a well-established approach (Broy 2003, Conrad and Fey 2005, Schäuffele and Zurawka 2005).

2.2.3. Model-Based Testing in the Automotive Industry
In terms of definitions, this thesis will adopt the model-based testing (MBT) definition presented by Utting et al. (2006), which describes MBT as “… a variant of testing that relies on explicit behavior models that encode the intended behavior of a system and possibly the behavior of its environment. Pairs of input and output of the model of the implementation are interpreted as test cases for this implementation: the output of the model is the expected output of the system under test (SUT)”.

45
A MBT approach typically relies on a specification-based model of the SUT. This implies that the developer needs to know the inputs and outputs of the system as well as the restrictions on these input-output pairs. The model is used to generate tests according to some previously specified test generation and selection criteria. During the process, the model itself evolves from an initial logical model to a concrete implementation model and subsequently into written code (Conrad and Fey 2005). Reviews on MBT can be found in (Broy et al. 2005, Utting 2005, Utting and Legeard 2006, Utting et al. 2006).

MBT offers an inexpensive way of testing complex systems before the physical system has been built and its early use has the potential of finding errors when they are cheaper to rectify. Literature reports various benefits of MBT, for example, a survey (Binder 2012) conducted by Robert Binder in late 2011, reports that MBT reduces escaped bugs by 59%, it reduces costs by 17% and reduces testing duration by 25%.

Developing automotive software commonly requires cooperation and synchronization among a number of companies and at different steps of the development and testing process. If we revisit the concept the V-Model with a focus on testing activities, the MBT process in the automotive industry can be depicted as in the Figure 5 below.

---

11 The survey targeted mainly attendees of the Model-Based Testing User Conference organised by the European Telecommunications Standards Institute (ETSI) and the Fraunhofer-Institutes FIRST and FOKUS in Berlin (October 18-20). One could speculate that the survey participants had a positive bias towards MBT. That said, the survey was conducted under the auspices of the conference and its results were published online. Comments to the findings of the survey were left open to the community in an online forum and blog.
Testing activities in different stages of the V-Model depicted above, among other things, differ from the opacity of the testing technique. When testing the function code, white-box techniques such as checking inside the structure of the program, testing the program branches, paths, conditions, etc. could be more appropriate. When moving up the right tail of the “V”, testing techniques move into grey and black-box ones, with conformance testing, which could be seen as a special case of acceptance testing, almost always done in a black-box approach.

To date, there is considerable research gone into MBT (Utting et al. 2006) and its benefits (Pretschner et al. 2005). The past years have seen different flavours of Model Based Testing and Utting et al. (2006) have proposed a taxonomy, which categorizes them into seven different dimensions as depicted in Figure 6.
In the taxonomy diagram above, vertical arrows denote a continuous range of possibilities, options divided by a forward slash denote mutually exclusive alternatives and continuous curved lines denote alternatives that are not necessarily mutually exclusive. Utting’s taxonomy provides a fairly comprehensive picture of the model-based testing realm at a glance. This taxonomy is revisited in Chapter 4 (Section 4.3.3) of this thesis and some changes to it are proposed.

From a theoretical point of view, two important issues in model-based testing are model redundancy and the level of abstraction in a model (Utting et al. 2006). Model redundancy deals with the choice between building only one model, which will be used both to test and to implement the SUT, or building an additional model to be used exclusively for testing. In terms of the level of abstraction in the model, the tester faces the nontrivial task of having a clear model that brings out the levels of abstraction (Pretschner et
al. 2007) and at the same time conveys a clear understanding of the omitted details and whether they can be tested via the said model (Utting et al. 2010).

While Utting et al. (2006) make a case for the need for model redundancy, Conrad and Fey (2005), abide by the automotive state-of-practice to build a single model for testing, which then evolves into an implementation model. The main argument behind the automotive practice of using an existing functional model of the system as the actual model for testing remains an economical one. While in principle the idea of having separate models just for testing has many benefits, the situation in practice remains that in addition to the initial effort for building a separate model for testing, the maintenance cost of such model can be considered prohibitively expensive by many companies in the field. There clearly is a need for domain specific studies that examine the economics of having a separate model for testing. A framework, which discusses such economics while taking in consideration factors such as model size, model complexity and expected change in requirements would be beneficial for the industry.
2.3. A Review of a Selection of Testing Toolsets and Testing Techniques

Despite the affinity of model-based testing with the automotive industry, it, still tends to be done in a bespoke and non-systematic fashion (Lamberg et al. 2004). Most of the research on MBT (Utting et al. 2006) comes short of addressing the domain specificity of automotive software and while a few exceptions exist (Conrad and Krupp 2006, Zander-Nowicka 2009), easy to use, high quality, formalised, model-based testing methodologies are in short supply.

Overall, most of the proposed methods impose restrictions which can prove too unrealistic for the complexity of real world systems. On the other side, while a number of standards do recommend formality, they confine themselves into mentioning a wide spectrum of techniques but do not elaborate on the specific criteria that would qualify a method or notation to reach the desired ‘formality benchmark’. Since there is a considerable amount of toolsets and approaches targeted at the embedded software field, which incorporate some degree of formality, it is of interest to have a closer inspection of the specific type of formality they incorporate. Hence a study was taken of 20 tools (Table 3) that are used in state-based systems, which have been developed between the years 1991-2006 and incorporate some degree of formality.

The tools are analysed across three dimensions: on their approach for test generation, on the type of formality they use and on the domain where they are used.

The test generation approach of each tool is further analysed from the test-selection criteria prospective, the type of coverage and whether the test approach is reactive. Also the test generation capabilities are analysed in terms of the test generation technology used, i.e. on whether the tests are generated manually or automatically; online or offline.
Each of the tools is reviewed briefly and below that narrative, based on the dimensions described above, each tool is analysed in tabular form.

While, based on the above, the analysed toolsets and methods can be organised according to a number of criteria, it was concluded that any one of these criteria was a bit forced for at least a few of the methods. Hence, for ease of reference it was chosen to discuss them ordered alphabetically.

The table below (Table 3) shows the toolsets and approaches considered in this review and their respective section numbers.

<table>
<thead>
<tr>
<th>Heading number</th>
<th>Name of the toolset</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>AGEDIS</td>
</tr>
<tr>
<td>2.3.2</td>
<td>ASML Test Tool</td>
</tr>
<tr>
<td>2.3.3</td>
<td>AutoFOCUS</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Conformance Kit</td>
</tr>
<tr>
<td>2.3.5</td>
<td>CTM and CTM_EMB</td>
</tr>
<tr>
<td>2.3.6</td>
<td>DOORS</td>
</tr>
<tr>
<td>2.3.7</td>
<td>GATel</td>
</tr>
<tr>
<td>2.3.8</td>
<td>LTG</td>
</tr>
<tr>
<td>2.3.9</td>
<td>Lurette</td>
</tr>
<tr>
<td>2.3.10</td>
<td>Lustre</td>
</tr>
<tr>
<td>2.3.11</td>
<td>Lutess</td>
</tr>
<tr>
<td>2.3.12</td>
<td>MATLAB/Simulink/Stateflow</td>
</tr>
<tr>
<td>2.3.13</td>
<td>PHACT</td>
</tr>
<tr>
<td>2.3.14</td>
<td>SCADE</td>
</tr>
<tr>
<td>2.3.15</td>
<td>STG</td>
</tr>
<tr>
<td>2.3.16</td>
<td>TestComposer</td>
</tr>
<tr>
<td>2.3.17</td>
<td>TGV</td>
</tr>
<tr>
<td>2.3.18</td>
<td>TorX</td>
</tr>
<tr>
<td>2.3.19</td>
<td>TTCN-3</td>
</tr>
<tr>
<td>2.3.20</td>
<td>TVEDA</td>
</tr>
</tbody>
</table>

### 2.3.1. AGEDIS

AGEDIS (Automated Generation and Execution of test suites for DIStributed component-based Software) (AGEDIS) was a research project (October 2000 to the end of 2003) of a consortium of academic and industrial research groups led by the IBM Research Laboratory in Haifa. The project aimed to develop methods and tools for automated software...
testing with a specific focus on distributed component-based software systems.

The AGEDIS tools are written in Java. It is designed based on an open architecture concept, and it provides stereotype interfaces between the SUT and its environment. The test architecture, which includes the mappings of abstract parts of the specification to the SUT is done via Extensible Markup Language (XML) schemas. It models the behaviour of the SUT via AML (Agedis Modelling Language), which is a Unified Modelling Language (UML) profile. The structure of the SUT is captured via class diagrams and IF (VERIMAG) is used as the action language.

AGEDIS uses TGV’s (see heading 2.3.17) test generation algorithms. It provides five different test-case generation strategies:

- Random test generation.
- Interface coverage, which aims to cover all the controllable and observable elements of the interface.
- Interface coverage with parameters, which in addition to the above aims to cover all parameter combinations.
- Specification based state coverage.
- Specification based transition coverage.

AGEDIS also provides two tools for tests analysis. One tool provides coverage analysis of data value combinations and method calls as well as generates test cases to cover the missed combinations. The other tool analyses the defects that have been found by clustering traces, which generate the same defect and condensing them into one defect producing trace.

Despite the initial promise, one of the major drawbacks of AGEDIS is that the tools were not made available outside of the project and they are not even obtainable for academic research use. On the plus side AGEDIS’ use of UML bears a kinship with the concept behind U2TP (UML Testing Profile) and has the potential to be adopted by the UML community for
testing. Furthermore, its open interface architecture and its use of XML have the potential of appealing to a relatively large number of testers.

Table 4: AGEDIS

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>It incorporates five test generation strategies: random test generation, state coverage, transition coverage, interface coverage and interface coverage with parameters. It can also generate test cases to cover the missed combinations of data values and method calls. Tests can be generated both online and offline. Test traces can be generated automatically as XML files. It also can execute test suites written in Java, C or C++.</td>
<td>It uses AML (Agedis Modelling Language), which is a UML profile. Uses IF as an action language. Provides some stereotypes interfaces for the communication of the SUT with environment.</td>
<td>It originated as an industrial and academic cooperation research project. Focused on distributed component-based software systems.</td>
</tr>
</tbody>
</table>

2.3.2. ASML Test Tool

ASML (Abstract State Machine Language) was developed by a Microsoft Research Group called Foundations of Software Engineering (Glässer et al. 2004, Grieskamp et al.). The tool is underpinned by the concept of Abstract State Machines (ASM), which are based on Gurevich’s work on Evolving Algebra (Gurevich 1995).

ASML expects the specification to be written in ASM form. This is transformed into a Finite State Machine (FSM) and a rural Chinese postman tour algorithm is used to generate the tests. The state-explosion problem is addressed by grouping states in equivalence classes and these groups are called hyperstates. The test cases are generated via state-based algorithms guided by a relevance condition, which decides if a state is relevant to be traversed. Test case generation needs guidance from the user and it is supported by a Graphical User Interface (GUI). The tool also has a parameter generator and gives the option to add filters that exclude states from being traversed (Gurevich 1995, Grieskamp et al. 2002, Grieskamp et al.). The equivalence relation is based on a set of Boolean conditions. States that cannot be distinguished via the set of Boolean conditions are placed in
one hyperstate. Even so, the number of FSM states can be too large and hence their number is restricted via a relevance condition, which decides if a state is worth traversing.

The translation from ASM into FSM can be quite complicated and furthermore the choice of both the equivalence relation and the relevance condition is based on the expertise of the tester. The above makes the tool relatively complicated to use.

Table 5: AsmL Test Tool

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tests are generated via a rural Chinese postman algorithm with some (manual) guidance from the user. The states are grouped in equivalence classes according to an equivalence condition and the number of traversed states is decided by relevance condition. The equivalence relation and relevance condition are designed manually.</td>
<td>It uses FSMs.</td>
<td>Microsoft research.</td>
</tr>
</tbody>
</table>

2.3.3. AutoFOCUS

AutoFOCUS (AutoFOCUS 2012, Huber et al. 1997) is a graphical modelling tool that can be used for code generation as well as for testing and verification. AutoFOCUS models a system via hierarchical time-synchronous Extended Finite State Machines (EFSMs). It offers three different methods of testing: based on a functional specification; based on structural specifications, which make use of the hierarchical nature of models in AutoFOCUS and based on statistical specifications. The models are translated into a Constraint Logic Programming (CLP) language and then executed symbolically. When generating the test sequences for a test specification, the specification is translated into CLP and is used to guide the test sequence generation; process, which includes the restriction of the search space and the decision of when to terminate the test sequence. The generated test sequences are symbolic test sequences and all the variables need to be instantiated. AutoFOCUS can also generate test cases, which conform to the required coverage criteria of the model.
The tool allows for the variables to be instantiated randomly or based on a limit analysis. Hence, if a systematic approach is not used when instantiating variables, the quality of tests could suffer. AutoFOCUS has been used in a number of case studies (Pretschner et al. 2001).

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offers testing based on functional, structural or statistical specifications.</td>
<td>Uses ESMS and CPL.</td>
<td>Distributed systems.</td>
</tr>
<tr>
<td>It can generate tests based on model coverage.</td>
<td></td>
<td>It has been used in a number of case studies.</td>
</tr>
<tr>
<td>It automatically generates symbolic test sequences. Test generation is guided via specification-based constraints.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The generated test sequences can be instantiated randomly or based on a limit analysis.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.4. Conformance Kit

Conformance Kit was developed by KPN Research in cooperation with the Dutch PTT for the automatic testing of protocols (Kwast et al. 1991). It expects the specification to be in the form of an EFSM. The specification can be split among different EFSMs, which can communicate through gates (a tool-specific concept, which regulates the communication among EFSMs). The tool allows refining communicating EFSMs into one EFSM and incorporates a converter, which translates EFSMs into FSMs.

Tests can be generated via a number of FSM algorithms, which include transition tours, unique input/output (UIO) sequences and generation of random sequences.

The strengths and restrictions of its test generation capabilities, among other things, depend on, which algorithm was followed\(^\text{12}\). After the tests are generated, they are captured in the syntactical notation of TTCN-3 (see 2.3.19).

\[^{12}\text{Please refer to chapter 3 in this thesis for a discussion of FSM test generation algorithms.}\]
The tool has been incorporated into a number of other tools including PHACT (see 2.3.13) and TVEDA (see 2.3.20). The limitations of the tool include the fact that it has not been made available publicly and that it cannot execute the tests it generates in TTCN.

Table 7: Conformance Kit

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test generation is performed via several FSM techniques. It can generate tests in TTCN’s syntactic notation but it cannot execute them. It can deliver some level of specification coverage.</td>
<td>Uses EFSMs and FSMs. Protocol testing. Conformance testing.</td>
<td></td>
</tr>
</tbody>
</table>

2.3.5. **CTM and CTM\textsubscript{EMB}**

The Classification Tree Method (CTM) is based in the category-partition method (Ostrand and Balcer 1988) and the concept of classification trees used to represent structured test cases (Grochtmann and Grimm 1993).

In the CTM, the input domain of the SUT is first partitioned in equivalence classes, which correspond to the different data sources. The partitions are called *classifications*. Each of these classifications then is further partitioned based on the possible range of their values. These inputs will form the top classifications in the tree (the top ‘leafs’ in the tree). The classification-tree defines the columns of a *combination table*. The test cases are built by cutting across the tree horizontally and combining different values from different partitions. The actual test scenarios are represented as the rows of the combination table.

The original CTM did not have an explicit concept of time. This was targeted by Conrad and Fey (1999), which proposed CTM\textsubscript{EMB}\textsuperscript{13}. CTM\textsubscript{EMB} provides a systematic approach to design test cases for embedded software based on functional specification test scenarios (test sequences). It provides the graphical notation to describe both time continuous test-scenarios

\textsuperscript{13} Initially the methodology was called CTM/ES. Since it is closely related to the work in this thesis, it will be discussed in more detail than some of the other tools and methods.
(Conrad et al. 1999) and event based test-scenarios (Conrad and Krupp 2006).

In CTM_{EMB}, each input of the SUT is represented as a classification in the tree. Each input domain is further partitioned into intervals of values. Each row in the combination table below the tree represents a test step and several test steps can form a test sequence (the steps are in temporal order and describe the course of the inputs overtime). Different graphical markings are provided to capture both time-continuous and event based test scenarios. CTM_{EMB} does not lend itself easily to capture reactive testing.

CTM_{EMB}, was developed in cooperation with Daimler Chrysler and it has been used in a number of automotive software development projects mainly for testing in-vehicle software in a model-based approach (Lamberg et al. 2004, Aldrich 2004). The methodology has been used in conjunction with the M/S/S environment.

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>It provides offline generation of test sequences.</td>
<td>Tabular notation and</td>
<td>Industrial.</td>
</tr>
<tr>
<td>It can provide data coverage,</td>
<td>graphical markings.</td>
<td>Embedded automotive</td>
</tr>
<tr>
<td>requirements coverage and test case</td>
<td></td>
<td>black-box testing</td>
</tr>
<tr>
<td>specification coverage.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does not offer reactive testing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uses simulation in the M/S/S environment.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.6.  DOORS

DOORS (Dynamic Object Oriented Requirements System) is a tool, which offers requirements management. It was originally developed by Telelogic and it is currently distributed by IBM as part of their Rational DOORS suite (IBM).

DOORS is not a testing tool but it is included here since it is widely used in the automotive industry to keep track of requirement coverage during the testing process. The tool can directly import Word and Excel documents and the requirements can be tagged and categorized. Requirements from
different documents and sources can be linked and the multiplicity of their relationship can be made visible.

DOORS is used in automotive, aerospace, medical and electronic and rail services.

**Table 9: DOORS**

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tool does not generate tests. It can keep track of specification coverage.</td>
<td>Requirement engineering. Used in industry.</td>
<td></td>
</tr>
</tbody>
</table>

### 2.3.7. GATel

GATel (Marre and Arnould 2000) can automatically generate test sequences from test objectives expressed in Lustre (*please refer to 2.3.10*). To generate its tests, GATeL needs the SUT or a complete specification for the SUT, an environment description and a test objective; all submitted in Lustre code. In difference from other Lustre-based tools such as Lutess (*section 2.3.11*) and Lurette (*section 2.3.9*) the test generation starts from the last state of the system. GATel finds a test sequence that satisfies a set of constraints, which include invariant properties and a test purpose. This is done via a CLP (it uses Prolog), searching the sequence backwards via a backtracking algorithm and the test sequence is generated offline, before the SUT is executed.

GATel also offers the possibility of domain splitting, in other words the possibility to zoom in a section of the domain, defined via the set of constraints, and split it recursively. The tool then attempts to produce a test sequence for each sub-domain. However since the tests are run backwards, the splitting of the domain can only affect the end of the test sequence (i.e. the initial constraint). GATeL allows both state invariance properties and state path predicates to be included in test sequence generation, albeit requiring every input to be provided in Lustre.
Table 10: GATel

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>It generates test automatically via a backward constraint propagation algorithm. When precise calculations are not possible, the backtracking algorithm needs to rely on some level of 'educated guess work from the tester. The tests are generated offline.</td>
<td>Uses Lustre and Prolog</td>
<td>It has been used in academic environments, the French Nuclear Certification Agency, as well as in avionics and automotive research centres.</td>
</tr>
</tbody>
</table>

2.3.8. LTG

Leiros Test Generator (LTG) started its life as the tool called BZ-TT. BZ-TT offered test-case generation based on formal models written in B, Z and Statecharts and it evolved into the commercial product LTG by Leirios. LTG expects the model and the test purpose to be presented as UML models. It provides a number of testing strategies: partition analysis, cause-effect testing, boundary value testing and domain testing. The tool uses a constraint solver to symbolically execute the formal input model and to generate abstract test cases. The generated abstract test cases then are translated into executable test scripts (Jaffuel and Legeard 2006).

The tool offers as test coverage criteria: multiple condition coverage, boundary coverage, and transition coverage. LTG has been applied to several case studies (Legeard 2003, Jaffuel and Legeard 2006).

Table 11: LTG

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tool uses a constraint solver to symbolically execute the formal input model and to generate abstract test cases. The generated abstract test cases then are translated into executable test scripts. LTG provides a number of testing strategies: partition analysis, cause-effect testing, boundary value testing and domain testing. It offers multiple-condition coverage, boundary coverage and transition coverage.</td>
<td>UML (Statecharts), B, Z.</td>
<td>Research and commercial</td>
</tr>
</tbody>
</table>
2.3.9. Lurette

Lurette is aimed primarily at testing reactive systems. It is based on the synchronous dataflow language Lustre (see 2.3.10) and it builds its test harness from the SUT, the oracle and an environment description. The SUT and the oracle need to be presented as a C-file while the environment description is in Lustre. It tests the SUT in a black-box fashion. The environment description is a synchronous observer, which evaluates to TRUE when it ‘observes’ a valid sequence of events in the environment (Raymond et al. 1998).

The test sequences are generated randomly (based on a uniform distribution). Test cases are generated step-by-step on-the-fly, i.e. for the initial state in each step the next output is computed, and this output is used to compute the input for the next state. To validate systems with numerical inputs and outputs, Lurette translates all numerical constraints into Boolean variables.

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>The test sequences are generated randomly (based on a uniform distribution). The SUT is tested as black-box. Test cases are generated step-by-step on-the-fly, i.e. for the initial state in each step the next output is computed, and this output is used to compute the input for the next state.</td>
<td>Based on the synchronous dataflow language Lustre</td>
<td>Academic. Testing of reactive systems</td>
</tr>
</tbody>
</table>

2.3.10. Lustre

Lustre is a dataflow oriented and time-synchronous programming language for reactive systems (Halbwachs et al. 1991, Caspi et al. 1987).

---

Test sequence generation (Bousquet et al. 1999) is based on an environment description written in Lustre, the last output from the SUT and a set of probabilistic constrains. In other words, it produces a test sequence, which when used as an input for the SUT, it will satisfy the environment' constraints.

The test sequence generation can also be guided by operational profiles, safety properties and chosen behavioural patterns.

Lustre only works with SUTs, which have no numerical inputs or outputs and it cannot generate tests based on coverage criteria.

This tool has been used in a number of industrial applications (Parissis and Ouabdesselam 1996, Ouabdesselam and Parissis 1995).

Table 13: Lustre

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides fully automated test sequence generation and execution.</td>
<td>Lustre</td>
<td>Synchronous reactive systems. Used in several industrial applications.</td>
</tr>
<tr>
<td>Test sequence generation can also be guided by operational profiles, safety properties and behavioural patterns.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>It cannot generate tests based on coverage criteria.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.11. Lutess

Lutess is a test environment for synchronous reactive systems. The tests are generated automatically based on environment description written in Lustre (Halbwachs et al. 1991). The test sequence generation is guided by probabilistic constraints, which help to predict if a test sequence is relevant. Lutess can provide random testing; testing based on operational profiles; testing based on safety properties to be tested as well as testing based on behavioural patterns to be executed (du Bousquet and Zuanon 1999, Belinfante et al. 2005).
Lutess only caters for Boolean variables (the test harness, cannot work with numerical inputs or outputs), and, it cannot generate tests based on coverage criteria.

It has been reportedly used in industrial applications (Ouabdesselam and Parissis 1995, Parissis and Ouabdesselam 1996).

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides fully automated test sequence generation and execution.</td>
<td>Lustre</td>
<td>Academic and industrial.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Used for synchronous reactive systems</td>
</tr>
<tr>
<td>It can provide random testing; testing based on operational profiles; testing based on safety properties to be tested as well as testing based on behavioural patterns to be executed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The test sequences are generated on the fly while the SUT is being executed.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.12. MATLAB/Simulink/Stateflow

MATLAB\textsuperscript{15} (matrix laboratory) is a high-level programming language and an interactive computing environment by MathWorks (MathWorks 2008). At its core, it operates by organizing all numerical information in a matrix form. MATLAB can be used for a vast range of applications, which include signal processing and communications, image and video processing, control systems and test measurement. Among other things, it allows implementing algorithms, creating user interfaces and to interface with other toolsets.

Simulink is tightly integrated with MATLAB and is a block diagram environment, which can be used in model-based design and simulation. It can talk in real time with MATLAB.

\textsuperscript{15} M/S/S is not strictly speaking simply a testing tool. It is included here due to this research’s interest in it.
Stateflow is one the environments that extends Simulink and it caters for simulating systems, which are captured as state machines and/or flow charts. The Stateflow state machines are implemented as Statecharts.

M/S/S is quite a complex environment and it is outside the scope of this document to present a thorough description of the M/S/S environment. The Statecharts as they are implemented in Stateflow, due to the interest they present for this research will be discussed in the next chapter. From the prospective of the focus of this research, is also interesting to note that the M/S/S environment can measure test coverage for a simulated SUT.

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>Mathematical notation, Statecharts</td>
<td>Industrial and academic</td>
</tr>
</tbody>
</table>

### 2.3.13. PHACT

PHACT (PHilips Automated Conformance Tester) is based on the Conformance Kit tool (2.3.4). In addition to the capabilities of the Conformance Kit, PHACT can execute the TTCN tests it generates.

The added functionality in PHACT has three components: the supervisor, the stimulator (which provides the inputs for the SUT) and the observer (which observes the outputs of the system). The supervisor, based on the observed behaviour, gives a pass or fail verdict to the executed TTCN test suite. The tool can also generate a test log, which can be translated\(^\text{16}\) into a Message Sequence Chart (MSC).

PHACT does not appear to provide reactive testing. It is not publically available but it has been made available to a number of research groups and it has been used to conduct a numbers of test studies including (Heerink et al. 2000). The literature also reports on comparative case studies where PHACT has been compared somehow negatively to other tools such as TGV

\(^\text{16}\) It can be translated into a Message Sequence Chart from the commercial tool SDT from Telelogic.
(2.3.17) and TorX (2.3.18). For example, according to (Goga 2001), PHACT, which uses UIO algorithms, has less detection power than TGV and TorX, when used in industrial practice.

Table 16: PHACT

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>In addition to the functionality of Conformance Kit (2.3.4), it can also execute test cases.</td>
<td>TTCN3</td>
<td>Refer to ConformanceKit (2.3.4)</td>
</tr>
</tbody>
</table>

### 2.3.14. SCADE

SCADE (Safety Critical Application Development Environment) is a tool suite for model-based development of safety-critical systems. It incorporates requirements analysis, modelling, design, implementation, and verification capabilities for critical embedded control applications. SCADE is used in many domains in industry including avionics, automotive, nuclear and train transportation systems.

SCADE is based on a graphical implementation of Lustre (8.9.10) and uses its formal semantics.

SCADE’s tools include a testing environment called SCADE QTE (Esterel Technologies), where, the tests can be specified in DOORS, ACCESS etc. SCADE allows to connect the tests to the specification and facilitates the generation of tests via spread sheets and mathematical based tools such as Scilab (The Scilab Consortium). The tool can also generate test conformance reports.

SCADE offers direct gateway products to the M/S/S environment, which is of particular interest for this research.
### Table 17: SCADE

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>The environment is underpinned by Lustre. Can use optimized spreadsheets to generate tests and it can use tools such as Scilab to specify tests. SCADE Can connect to tools such as DOORS, ACCESS, M/S/S.</td>
<td>Lustre</td>
<td>Industrial tool. Used in avionics, automotive, nuclear and trains.</td>
</tr>
</tbody>
</table>

#### 2.3.15. STG

STG (Symbolic Test Generator) (Clarke et al. 2002) is a symbolic test-case generator, which has incorporated the ideas behind TGV (see 2.3.17) and TorX (see 2.3.19), and, has added symbolic treatment for variables. STG implements a symbolic form of ioco (Tretmans 1996a) and provides symbolic test generation and test execution.

STG takes as inputs a specification and a test purpose both written in an Input Output Symbolic Transition System (IOSTS)\(^{17}\) form. It can also accept specifications and test purposes written in NTIF (Garavel and Lang 2002) since it can translate them automatically into IOSTS-s. STG generates symbolic test suites, which are also IOSTSs and are then instantiated during execution. The generated symbolic test suite is reactive and covers all the specification-based behaviour, which is described by the test purpose. STG uses OMEGA (Kelly et al. 1995) for its symbolic computations and this limits the allowed data types to arrays of integers and enumerations. The symbolic test cases are translated into a C++ program to be executed.

STG has been used to automatically generate executable test cases, in a number of case studies, which include the CEPS (Common Electronic Purse Specification) and the 3GPP (Third Generation Partnership Program) smart card (Clarke et al. 2001).

---

\(^{17}\) An IOLTS in essence is a LTS (Labelled Transition System) with its labels partitioned into three sets: the stimuli set, the observations set and the internal actions set.
Table 18: STG

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>It can automatically generate a symbolic test suite. The test cases are translated in C++ to be executed.</td>
<td>Uses IOSTS</td>
<td>Academic. Used in a number of case studies.</td>
</tr>
<tr>
<td>Expects as inputs a test specification and a test purpose, where the test purpose guides the test generation process.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The test suite is reactive.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3.16. TestComposer

TestComposer is based on TVEDA and TGV and it has been incorporated in the commercial tool ObjectGeode\(^\text{18}\).

TestComposer expects the specification to be written in SDL (Specification and Description Language), and generates the tests based on a set of test purposes, which in essence are sequences of input and output events exchanged between the SUT and its environment in a black-box fashion. Its test generation is based on finding paths in the SDL specification that correspond to the test purpose.

The tool can also ‘complete’ incomplete test purposes. It automatically computes the missing events based on a (specification based) state exploration algorithm. The same algorithm can also be aimed at achieving a specified system coverage.

It can adopt other test specification languages, including TTCN-3 (2.3.19), via API interfaces.

Table 19: TestComposer

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>It can automatically generate tests based on a path-finding algorithm. The algorithm can find paths in a specification written in SDL, such as to correspond to the test purpose</td>
<td>SDL</td>
<td>Incorporated in a commercial tool</td>
</tr>
<tr>
<td>It can deliver a specified percentage of model coverage.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^{18}\) It was originally developed by Verilog, which in December 1999 was acquired by Telelogic. Telelogic then was bought by IBM in 2008.
2.3.17. TGV

TGV is a test generator that implements the implementation relation $ioco$ (Tretmans 1996b). It serves as one of the test generation engines for TestComposer (2.3.16) and is the test generation engine for the Caesar Aldebaran Development Package (CADP) (Fernandez et al. 1996) and AGEDIS (see section 2.3.1). The underlying model of TGV is an Input Output Labelled Transition System (IOLTS). It takes as inputs the specification and the test purpose, which both need to be written as IOLTSs. The test purpose guides the test generation process. The generated test cases are IOLTSs that report the test verdict as ‘Pass’, ‘Fail’ or ‘Inconclusive’. The test suite then can be presented in TTCN or in one of the graph formats of CADP.

The test generation is done in steps, which are mostly performed on-the-fly. The on-the-fly approach followed is different from the ‘usual approach’ followed by tools such as Lutess (2.3.11), Lurette (2.3.9) and TorX (2.3.18). TGV follows what can be described as an on-demand approach on reverse, where the demands imposed on earlier steps depend on the later ones (aimed at minimising any work done in the earlier steps, which will not be needed by the later steps).

One of the limitations of TGV is that all the variables are instantiated for a large set of values and can produce a rather large amount of test cases.

TGV has been used in many case studies and has been interfaced with different specification languages including LOTOS (Fernandez et al. 1996), SDL, UML and IF.

![Table 20: TGV](image)

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests are generated automatically using a reverse on demand on-the-fly approach.</td>
<td>IOLTSs</td>
<td>Serves as the test generation engine for other tools. Academic. Testing of protocols.</td>
</tr>
<tr>
<td>Uses graph search algorithms.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The generated test cases are IOLTSs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.3.18. TorX

TorX presents itself as both an architecture for testing and a tool for test generation and execution (du Bousquet et al. 2000). The tool is based on the implementation of the ioco theory (Tretmans and Brinksma 2002) and in principle can handle models captured in any language as long as they can be converted into a Labelled Transition System (LTS). Tests can be generated in batch or on-the-fly mode. In the batch mode all the tests are generated and then executed in a second phase. The batch mode has not been fully implemented, but in this mode, the tool can execute test cases generated by TGV (8.9.17). In the on-the-fly mode the tests are generated on demand in a graded step-by-step approach. That is to say that one test step is generated then executed and then the next test step is generated, which is also executed etc. This brings the advantage that the state space exercised from a specific test run is restricted to a subset of the complete state space. In general, a random walk of the state space is done or the test process can be guided from a test purpose. The test purpose is expressed in a regular expression language called jararaca.

TorX provides an open interface (TorX Adapter interface), which needs to be implemented by the user in order to connect a SUT to it.

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests can be generated in batch or on-the-fly mode.</td>
<td>LTS</td>
<td>Academic, protocol testing</td>
</tr>
<tr>
<td>Test generation is done via a random walk of the state space or by guiding the walk in the state space via a test purpose.</td>
<td>Jararaca</td>
<td></td>
</tr>
<tr>
<td>The constrains are captured in a regular expression language called jararaca</td>
<td></td>
<td></td>
</tr>
<tr>
<td>It claims that it can any test models that can be converted into LTS.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.3.19. TTCN-3

Testing and Test Control Notation Version 3 (TTCN-3) is a standardized modular language designed specifically for testing. TTCN-3 is currently the third version of what started life as TTCN - the Tree and Tabular Notation (ITU 2006).

In its initial form, TTCN was a tabular notation used to describe tests. It had several shortcomings including the lack of ability to deal with concurrency in the system under test. This was addressed in the second version called TTCN-2, which could also ‘understand’ the concepts of module, package, encapsulation and had the ability to manipulate ASN.1 (Abstract Syntax Notation One) (ITU 2006).

The third version, which would be known as the TTCN-3, incorporates a generalised text-based test language called the TTCN-3 core language. Its syntax is defined in ETSI ES 201 873-1 (ETSI 2008a) and its operational semantics are defined in ETSI ES 201 873-4 (ETSI 2008b).

The core language can also be presented in a number of different formats, which include the Tabular Format for TTCN (TFT) and the Graphical Format for TTCN (GFT). Other standardized or even proprietary presentation formats are possible.

The TTCN-3 graphical presentation format, GFT, supports the presentation and the development of TTCN-3 test cases as Test Sequence Charts, which are derived from using Message Sequence Charts (MSCs) with test specific extensions. The TTCN-3 Core Language may be used independently of TFT and GFT. However, TFT and GFT cannot be used without the core language.

TTCN-3 is defined via a multipart standard, which covers the textual core TTCN-3 language, its different presentation formats, the execution interfaces as well as other language mappings to TTCN-3.

---

19 TTCN-3 is discussed in considerably more detail than the other approaches due to the fact that it has been a very active field of research in the last few years.
The TTCN-3 multipart standard:

- Part 1: "TTCN-3 Core Language";
- Part 2: "TTCN-3 Tabular presentation Format (TFT)";
- Part 3: "TTCN-3 Graphical presentation Format (GFT)";
- Part 4: "TTCN-3 Operational Semantics";
- Part 5: "TTCN-3 Runtime Interface (TRI)";
- Part 6: "TTCN-3 Control Interface (TCI)";
- Part 7: "Using ASN.1 with TTCN-3";
- Part 8: "The IDL to TTCN-3 Mapping";
- Part 9: "Using XML with TTCN-3";
- Part 10: "TTCN-3 Documentation Comment Specification";
- Part 11: "Using C with TTCN-3".

TTCN-3 can describe relations between test cases such as sequences, repetitions and dependencies on test outcomes, dynamic concurrent test configurations and testing for synchronous and synchronous environments. It has been specifically designed for black box testing and certification and it covers areas that include conformance and functionality testing; interoperability and integration testing; and, stress testing (Schieferdecker 2006).

TTCN-3 does not deal with the design methodology of tests and neither does it lend itself easily to test automation.

TTCN-3 to date is supported by a number of commercially available toolsets (Schieferdecker 2006).
Table 22: TTCN-3

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTCN-3 is a standardized language designed specifically for testing. It supports the development of test cases as Test Sequence Charts (TSC), which are derived from using Message Sequence Charts (MSCs) with test specific extensions.</td>
<td>Uses its proprietary notation. The core language, in addition to the text-based format, provides a tabular and a graphical format. Uses Test Sequence Charts and supports MSCs, XML and C.</td>
<td>Designed for black box testing and certification. It covers areas that include: regression testing; conformance and functionality testing; interoperability and integration testing, and load/ stress testing.</td>
</tr>
</tbody>
</table>

2.3.20. TVEDA

TVEDA (Clatin et al. 1996) was developed by the R&D centre of France Telecom, aiming to deliver automatic conformance testing for protocols. The tool expects the specification to be in Estelle or SDL. Test generation is based on testing ESFM transitions and TVEDA can generate transition tours or single tests for every transition. TVEDA attempts to achieve complete transition coverage and to mitigate the problem of state explosion it uses symbolic execution or reachability analysis (to find feasible paths).

Its strong points include the fact that it has been used to tests protocols and can generate output in the TTCN format. Furthermore, parts of the TVEDA have been incorporated into the TestComposer tool, which on its turn was incorporated into the commercial tool ObjectGeode from Telelogic\textsuperscript{20}.

\textsuperscript{20} Telelogic AB was a Swedish company bought from IBM in April 2008 and now is part of the IBM software group.
Table 23: TVEDA

<table>
<thead>
<tr>
<th>Test Generation</th>
<th>Type of Formality Used</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test generation is based on testing of ESFM transitions. It can generate transition tours or single tests for every transition. It has two strategies to compute feasible paths: symbolic execution and reachability analysis. It aims to deliver transition coverage. It expects the specification to be in Estelle or SDL language.</td>
<td>Uses EFSMs and SDL. It was not initially based on formal theory but on empirical principles. Most of the empirical principles have been validated theoretically after the tool’s development</td>
<td>France telecom, conformance testing protocols</td>
</tr>
</tbody>
</table>
2.4. Reflections on the Reviewed Tools

Most of the reviewed tools have a strong formal aspect and they aim to deliver automatic test generation, even so, they only deliver the latter partially. The automaticity of test generation is achieved at varying degrees and the need for involvement from human testers remains strong. Overall, achieving a completely automatic on-the-fly test generation remains an eluding goal as yet to be achieved.

Likewise, very few of the toolsets can cater for a fully reactive testing process. Toolsets and approaches that can choose the next test sequence based on the last observed behaviour of the SUT remain elusive or quite restricted in their reactive capabilities.

From a perspective of the ‘preferred’ formal notations to be incorporated in tools, Figure 7 shows that there is a cluster of tools, which use FSMs and EFSMs (respectively 15% and 20%)\(^{21}\).

![Figure 7: The type of formality or notation used in tools](image)

FSMs and EFSMs also result to be the most popular in industrial tools (Fig. 8). IOLTS on the other side do not seem to have breached the gap between

\(^{21}\) The same graph also shows that 25% of the tools under review use Lustre. This is not a ‘safe’ value to draw conclusions on since this percentage is skewed higher by the sample of the tools. Five out of the twenty tools that were reviewed were based on Lustre. Hence, the relevance of this percentage is quite limited.
academia and industry. Whereas, Statecharts appear to be the most popular approach in tools, which are used by both academia and industry.

Interestingly, while a considerable 50% of the reviewed tools have been used in industry, only 20% of them are commonly used both in industry and academia (Fig. 9).
Finally, in view of the quality assurance demands on the required specific coverage to be achieved by testing, the tools were analysed in terms of the type of coverage they provide (Fig. 10). The majority of tools (60%) offer some degree of structural model coverage, reflecting among other things, the percentage of the tools, which are targeted to the entire software testing process (and not only to the strictly black-box approach of the conformance testing scenario). Stochastic coverage is incorporated in 20% of the tools. This, perhaps partly reflecting the relative ease of programming stochastic approaches.

![Figure 10: Test coverage delivered by different tools](image)

### 2.5. Conclusions

This chapter establishes some of the definitions that will be referred to when in need to establish the intended meaning for some concepts, which are relevant for this research.

The chapter discusses some of the specific issues faced by the automotive embedded software with a focus on the issues surrounding model-based testing of this type of software.

Model-based testing is a favourite approach in the automotive industry, yet, approaches that are targeted specifically to the automotive embedded
software are needed. Among the features that these approaches need to have are the incorporation of automatic test generation and test reactivity.

Based on the QA requirements for the need to incorporate formality in the testing process, twenty toolsets and approaches that incorporate formality were discussed and analysed against the backdrop of test generation, the type of formality used, and the domain they are used in.

Overall, achieving a completely automatic on-the-fly test generation remains an eluding goal as yet to be achieved. FSMs and EFSMs result to be the most popular formal notation, which has been incorporated in industrial tools, whereas, Statecharts appear to be the most popular notation in tools used by both academia and industry.

Stochastic test generation features quite strongly among the tools, making the case for approaches that can deliver a systematic underpinning of test generation.

The next chapter discusses a selection of quality assurance software standards, which are relevant for embedded software as well as the state of practice in formalized development.
3. Safety Standards and Formalized Development

“The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.”


This chapter discusses some of the most relevant QA standards for automotive embedded software. Due to the standards’ requirements for formality, the chapter reviews formal testing and selection of testing techniques. The chapter pays particular interest to Statecharts and how they have been implemented in the Stateflow environment. It concludes with reflections on the limitations of the reviewed techniques.

3.1. Standards

Software is very often used in safety-related applications such as medical applications, control of nuclear plants, avionics and automotive applications to name but a few.

This class of software must conform to the state of art in terms of scientific and technological maturity. Prevention of product liability claims requires, at minimum, adherence to the relevant guidelines and certification standards. For electrical and electronic systems, one of the main points of reference is the IEC 61508 (IEC 1998) standard for functional safety of electrical, electronic and programmable electronic safety related systems. Most of the major industrial sectors have developed their own standards based on the IEC 61508. For example, the nuclear sector has adapted the CEI 61513 (IEC 2001b) standard, the CEI 61511 (IEC 2003b) standard is adapted for industrial processes, the EN 50126/EN 50128/EN 50129 (IEC 1999, IEC 2001a, IEC 2003a) are adapted for the railway sector, the CEI
62061 (IEC 2005) standard is adapted for the safety of machines, the IEC 62304 (IEC 2006) standard is adapted for medical device software, DO-178B (RTCA 1992) and its revised version DO-178C (RTCA 2011) are adapted for avionics software and the new ISO 26262 (ISO 2011) is adapted for the automotive industry. While AUTOSAR (2013) is a relatively new automotive standard, which addresses the need for automotive software standardization.

From the above, IEC 61508, DO-178B and its revised version DO-178C, ISO 26262 and AUTOSAR are the most relevant for this thesis and hence are discussed in the next sections. The MISRA guidelines (MISRA 1994), which apply mainly to the UK market are also discussed due to their geographic relevance.

### 3.1.1. IEC 61508

IEC 61508 (IEC 1998) is based on the rationale that risk can never be completely eliminated but risk, which is non-tolerable must be eliminated. The standard addresses the complete safety life cycle and requires that safety must be considered from the beginning.

IEC 61508 (IEC 1998) defines the degree of rigour required in the software development process based on the concept of the Safety Integrity Level (SIL). The standard specifies the requirements for systems to be designed, implemented, operated and maintained to provide a SIL that varies from 1 to 4, where SIL 1 denotes the highest possibility of failure on demand (PFD) (0.1-0.01) and SIL 4 the lowest PFD (0.0001 – 0.00001) (Fig. 11).

<table>
<thead>
<tr>
<th>SIL</th>
<th>PFD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1 – 0.01</td>
</tr>
<tr>
<td>2</td>
<td>0.01 – 0.001</td>
</tr>
<tr>
<td>3</td>
<td>0.001 – 0.0001</td>
</tr>
<tr>
<td>4</td>
<td>0.0001 – 0.00001</td>
</tr>
</tbody>
</table>

**Figure 11: Safety Integrity Levels**

The SIL concept is defined as a “… discrete level for specifying the safety integrity requirements of the safety functions to be allocated to the safety-
related systems, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest” (IEC 1998).

IEC 61508 (IEC 1998) has a generic approach, which covers a wide range of sectors and provides flexibility in adopting a broad range of techniques and measures to achieve functional safety. This includes flexibility into adopting new and improved methods. This generic approach is also one of the limitations of the standard since it needs interpretation to be used in a specific sector.

Other strong points of this standard are its worldwide acceptance and the wide range of domain-specific toolsets that support it.

One of shortcomings of the IEC 61508 (IEC 1998), a shortcoming, which is shared by other standards, is its opting to provide only qualitative measurements by which to measure system integrity. In other words, while it proposes safety integrity levels in the range of ten to the power of minus five, the proposed measurements for certifying software for a specific SIL are only qualitative.

In order to deliver the higher SILs, the standard calls for, but stops short of mandating, the use of formalised development of software. However, the standard limits itself into mentioning an overarching range of formal notations and formal methods that can be used in software development and stops well short of pointing to a process in which these methods could be incorporated.

3.1.2. DO-178B and DO-178C
The automotive industry has an invested interest into incorporating technologies such as X-by-Wire\textsuperscript{22} into cars. They often point to similar

\textsuperscript{22} X-by-Wire is a collective term for a number of by-Wire (Steer-by-Wire, Brake-by-Wire, etc) systems in the automotive industry, where the traditional mechanical control systems such as per instance the mechanical connection between the steering wheel and the wheels of the car, are replaced with electronic control systems, which translate the driver’s actions into electronic messages that are transmitted via in-vehicle networks to electromechanical actuators.
technology having been used successfully for years in the avionics, to support the argument that X-by-Wire can be safely deployed in cars. In view of this, DO-178B and its revised version DO-178C, which are standards for avionic software are briefly discussed here.

DO-178B was developed by RTCA (Radio Technical Commission for Aeronautics) in 1992. It incorporates quality criteria and guidelines for the development process of software in airborne systems. It defines five safety levels (quoting from the standard in Text Box 1):

- **Level A - Catastrophic**: Failure may cause a crash.
- **Level B - Hazardous**: Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the plane due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
- **Level C - Major**: Failure is significant, but has a smaller impact than a hazardous failure (for example, leads to passenger discomfort rather than injuries).
- **Level D - Minor**: Failure is noticeable, but has a smaller impact than a major failure (for example, causes passenger inconvenience or a routine flight plan change).
- **Level E - No Effect**: Failure has no impact on safety, aircraft operation, or crew workload.

DO-178B discusses requirements-based testing approaches and requires specific coverage metrics for specific safety levels. For the highest safety level (Level A), DO-178B requires the fulfilment of the MC/DC metric.

One, among the different calls for clarification of the definitions of the DO-178B, was to clarify the concept of verification in the model-based development context; and whether formal methods and model simulation could replace some or all of software testing.

In January 2012, a new version of this standard, the DO-178C/ED-12C, was made available in the public domain.

One point of interest for this research is DO-178C’s interest into the rapport of formal methods, model-based development and testing. One of the accompanying documents of the standard (called document DO-333)
addresses the use of formal methods so as to complement but not replace testing.

3.1.3. MISRA – Development Guidelines for Vehicle Based Software

MISRA (Motor Industry Research Association) is a UK based association and its guidelines are relevant mainly for the automotive industry in the UK. MISRA also uses a SIL concept but its meaning is different from that of IEC 61508.

MISRA’s SILs are in the range of 0-4, and are qualitatively defined combining the concepts of ‘controllability’ (i.e. the degree to which the typical user would be able to control the outcome or consequences of the a possible functional failure) and of ‘acceptable failure rate’ (i.e. how likely it is acceptable for the failure to occur) (MISRA 1994).

In difference from IEC 61508, MISRA only provides guidelines for the UK market and its existing tools are only used by the automotive industry. Furthermore it does not provide a route to certification and describes no methods or techniques to be adopted in order to achieve a specific SIL.

3.1.4. ISO 26262

ISO 26262 (ISO 2011), which was first published in the public domain in 2011, is the adaptation of IEC 61508 for the functional safety compliance of electrical and/or electronic (E/E) safety-related systems within road vehicles.

It defines the safety requirements that must be fulfilled by the design and development of the system for the entire automotive production process and it is the standard, which is more closely related to this research.

ISO 26262 addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems, including the interaction of these systems but it does not address physical or environmental hazards such as fire, heat, heat, fire, etc.

---

23 It applies to E/E systems installed in passenger cars with a maximum vehicle mass of 3500 kg, excluding special purpose vehicles such as the ones designed for drivers with disabilities.
electric shock, radiation, toxicity etc., unless they have been directly caused by some malfunction of E/E safety-related systems.

One of the key concepts of ISO 26262 is the ASIL (Automotive Safety Integrity Level), which in essence is a risk class associated with the consequences of a software function’s possible malfunctioning behaviour. The ASIL is determined at the beginning of the development process and is software based; it is calculated for a safety related software function and not for a hardware component.

![Figure 12: Key concepts of ISO 26262](image)

Each functional safety function is assigned an ASIL of A, B, C, or D, with D representing the most safety-critical processes and stringent testing requirements. The standard also defines a QM\textsuperscript{24} value, which denotes a non safety-critical function.

The focus of the ASIL is the mitigation of the possible harm that can be caused to the driver and other road users. It provides a risk assessment procedure, which determines the ASILs based on a combination of the

\begin{itemize}
  \item \textit{Exposure Time}: temporal window, characterized by specific operational situation (traffic, vehicle speed, split...) in which if the malfunction happens there is an hazardous situation
  \item \textit{Controllability}: avoidance of a specified harm or damage through timely reactions of the persons involved
  \item \textit{Severity}: measure of injury of the person involved in the hazard
\end{itemize}

24 The rating "QM" indicates that the observance of any established quality management system is sufficient to achieve the corresponding safety goal and that no additional requirements need to be taken from ISO 26262.
probability of *exposure time*, the possible *controllability* by a driver, and the outcome’s *severity* if a malfunction occurs (Fig. 12).

A hardware and/or software component of a system, which includes a number of safety functions with different ASILs, will typically\(^{25}\) inherit the highest ASIL of its functions.

The recommendations for software testing as part of the standard for software development are defined in the *Part 6* of the ISO 26262 standard. The recommendations include the minimum testing requirements based on the ASIL of the component and a number of recommended test approaches such as requirement-based tests, partition tests and structural coverage tests.

ISO26262 (ISO 2011) recommends that formal notations are used for the higher ASILs throughout the development process including specification, design, architecture and testing. For example *Figure 13* depicts the standard’s recommendations for the methods to be used for the verification of requirements for the different ASILs.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal verification by walkthrough</td>
<td>A</td>
<td>++</td>
</tr>
<tr>
<td>Informal verification by inspection</td>
<td>B</td>
<td>+</td>
</tr>
<tr>
<td>Semi-formal verification</td>
<td>C</td>
<td>+</td>
</tr>
<tr>
<td>Formal verification</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

\(^{25}\) Under specific circumstances, ISO 26262 allows the developer to lower the ASIL of a component. In other words if a component is composed from a number of functions, and one of them for example requires an ASIL of D, a company can use a technique defined as ‘*ASIL decomposition*’, to develop the component using the less stringent requirements of a lower ASIL. Producing a component according to the requirements of a lower ASIL, would at a first glance point to lower production costs. That said, a closer analysis of the ASIL decomposition technique points to the fact that in some cases the person-hours and the high level of expertise required, might offset the expected cost savings. To date, I am not aware of any studies, which analyse the cost benefits of the technique or any influence this technique might have on the overall quality and safety of a component.
The formal verification recommended in the standard is defined as an activity, which is used to ensure the correctness of the system under test as defined in the formal specification of its behaviour. This can be construed as a recommendation to formally specify the requirements, which can serve as the basis for any potential verification activities.

The ISO standard also introduces the use of specific coverage metrics for certain ASILs (as depicted in Fig. 14).

The rationale behind the requirement of specific coverage requirements is linked to the idea of completeness of test cases and avoidance of unintended functionality.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement coverage</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>MC/DC coverage</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

“++” The method is recommended for this ASIL
“+++” The method is highly recommended for this ASIL

Figure 14: Structural coverage metrics recommended by ISO 26262 (Part 6)

3.1.5. AUTOSAR

The automotive industry has a history of every player developing its own proprietary automotive software and of high confidentiality surrounding it all. A large consortium of automotive companies developed AUTOSAR (Automotive Open System Architecture) under the principle of “cooperate on standards, compete on implementation”. AUTOSAR’s main focus is to address the industry’s need for customization and common solutions, and, to address modularity, scalability, transferability and re-usability of functions.

AUTOSAR uses a component based software infrastructure, which is composed of application software components linked via a conceptual virtual function bus. All the application software components have
standardized interfaces defined in the standard, while the functionality they provide is up to the developer.

The different components can connect directly to the virtual bus via standardized interfaces due to AUTOSAR’s layered architecture approach (Fig. 15).

![AUTOSAR's Layered Software Architecture](image)

**Figure 15: AUTOSAR’s Layered Software Architecture (AUTOSAR development partnership 2013)**

In this layered architecture approach, the *Application Layer*, contains the functional applications of an ECU network. The AUTOSAR standard refers to them as *AUTOSAR Software Components*. The software architecture in this layer is component-based, composed of non standardized application software components that interact via standardized interfaces with the runtime environment layer. All the actual functionality provided by the applications is located in this layer.

The *AUTOSAR Runtime Environment (RTE)* layer provides all the communication services for the application software, i.e. all the *Software Components* communicate with other components and services, which could be located in the same ECU or not via the *RTE*. This is the only layer that
has a component-style architecture. Starting from the RTE layer and below, the software changes from component to layered style architecture. The task of the RTE is to make the AUTOSAR Software Components independent from the mapping to a specific ECU. The RTE is generated individually for each ECU and its implementation is both application and ECU specific, while its upper interface (the interface to the application layer) is completely ECU independent.

The Services Layer is the highest layer of the Basic Software and while access to I/O signals is covered by the ECU Abstraction Layer, the Services Layer offers: operating system functionality; vehicle network communication and management services; memory services management; diagnostic services and ECU state management. Its implementation is partly dependent on the microcontroller and it is ECU hardware and application specific. Its upper interface is microcontroller and ECU independent.

The ECU Abstractions Layer is where an abstracted design of the ECU resides. It offers an API (Application Programming Interface) to access devices and peripherals regardless of their location and the type of interface they have with the microcontroller. Its implementation is ECU dependent but microcontroller independent. At its upper interface (to the layer above) is both ECU and microcontroller independent.

The Microcontroller Abstraction Layer contains all the abstractions and peripherals of the microcontrollers. Its implementation is dependent on the microcontroller but its upper interface is microcontroller-independent.

This layered architecture is used on every ECU developed according to the AUTOSAR standard. The rationale behind this is to allow the software developers to use any number of software components without the need of knowing on which ECUs the components are installed or what type of hardware they are connected to. It is the Runtime Environment that has to make sure that the software components can communicate with one another
or with the hardware no matter on which ECU they are deployed.

In all matters of quality assurance, AUTOSAR abides by the requirements posed by ISO 26262. Since its release 4.0, AUTOSAR has also elucidated a number of previously not-specified safety-relevant issues to clarify its compliance with ISO 26262.

The standard itself, while benefiting from the participation of a large numbers of important players in the automotive industry, it has also attracted criticism for being too large as a result of including elements that would benefit particular members of the consortium. The standardization requirements have also attracted criticism on the fact that increases the cost of the processors. Whereas the supporters of AUTOSAR argue that switching from proprietary software to standardized software architecture offers cost and capacity benefits and furthermore standardization increases the degree of reuse.

In reality, the release of AUTOSAR has coincided with an unsettled economic climate in most markets. This has made it difficult to measure its real economic impact.
3.2. Formal Testing

During the course of the last four (plus) decades, the use of formal methods in improving the development process for safety-critical systems has been extensively researched (Bowen and Hinchey 1995b, Bowen and Hinchey 1995a, Bowen and Hinchey 2006, Bicarregui et al. 2009), and evidence of their benefits for industry has been analysed and documented (Bowen and Hinchey 1997, Chechik and Wong 2002, Wassyng and Lawford 2003). Yet, despite the evidence to support the benefits and need for formal methods, transfer to industrial practice remains slow and unsatisfactory (Broy 2009). Analyses of the reasons behind the slow uptake from industry point towards the need for powerful tools and pragmatic approaches to using these tools within the software process (Dill and Rushby 1996, Holloway and Butler 1996, Broy 2009), combined with standards and education (Hinchey 2003, Parnas 2010).

Real and perceived complexity, poor understandability, low readability and reviewability are among the issues that continue to affect the use of formal techniques. This, appears to corroborate the argument that if engineers and other stakeholders are not able to properly understand the techniques used in development, those techniques are not useful in practice (Broy 2009). In this regard, graphical notations tend to fare better and some researchers even dare to classify them as popular (Broy 2009).

The ‘rapport’ between formal methods and testing has had difficult beginnings. Research in formal methods initially was primarily aimed at verifying and proving correctness, while testing per definition, primarily aimed at showing the presence of errors. In the last ten years, there has been an increased interest in the use of formal methods in software testing, which recognises the complementary capabilities of formal methods and testing (Brinksma and Tretmans 2001).

The behaviour of an embedded system can often be abstracted as: it receives input from the environment (typically through sensors); it performs actions
(typically through actuators); the actions, which are performed typically depend on the state the embedded system is in; and the system’s state can change as a consequence of the received input or of the performed action.

Since this type of state-based behaviour can be captured easily via Finite State Machines (FSMs) (Text Box 2), they have been used for quite some time to both model and test these systems.

Defined formally, a FSM is a six-tuple \((X, Y, Q, q_0, \delta, O)\), where

- \(X\) is a finite set of input symbols also known as the input alphabet.
- \(Y\) is a finite set of output symbols also known as the output alphabet.
- \(Q\) is a finite set of states.
- \(q_0 \in Q\) is the initial state.
- \(\delta: Q \times X \rightarrow Q\) is a next state or state transition function.
- \(O: Q \times X \rightarrow Y\) is the output function.

For deterministic FSMS, for any state \(q\) in \(Q\) there is at most one next state.

Text Box: 2

Conceptually, the typical testing process of a FSM can be described as the process of designing and generating the right sequence/s of inputs for the machine and from the observed outputs and overall behaviour deduce if the FSM conforms to its specification.

Typically the testing problem assumes that there exists a formal specification for the system captured via a FSM and the tester needs to test the implementation of said specification (Hierons et al. 2009). The latter is called implementation under test (IUT). The tester has the transition diagram of the specification FSM and can observe the IUT’s input and output (I/O) behaviour.

26 In this thesis an IUT is equivalent to the SUT. In this chapter is kept the term IUT first to acknowledge the fact that there can be a model used to implement the system and another model used to test the system and second to keep with the terminology of this area (formal testing).
The underlying assumption of finite-state machine based testing methods (Chow 1978, Fujiwara et al. 1991) is to consider the equivalence in behaviour of state machines as a conformance relation and, based on this assumption, to attempt and produce a finite number of tests to check the equivalence in behaviour between the specification FSM and the IUT. Under this assumption, the process of testing whether the IUT conforms to the specification is called conformance testing.

An intuitive starting point for researchers was to harness the synergy between test generation and model checking. There is considerable research work in test generation derived by model checking techniques. Examples of these include (Jéron and Morel 1999, Ammann et al. 1998); Callahan and colleagues’ method (1996) on specification-based temporal formulas; Engels and colleagues’ model checking based test generation for intelligent networks (1997); and Gargantini and Heitmeyer’s (1999) test generation based on model checking while taking in consideration a number of control flow criteria.

In order to simplify the testing process, all of the proposed techniques (some of them not explicitly) are based on a fault model, which simplifies the testing problem by focusing on a small number of fault types that the IUT could have.

A widely used fault model for FSMs incorporates four fault categories. According to this model, for an implementation under test $I: (X, Y, Q, q_0, \delta, O)$ and a specification $F: (M, N, S, s_0, t, U)$, the considered fault categories are as described in Text Box 3.
1. **Operation error:** where fall any errors in the output. Defined formally, an operation error implies that for some state $q_i \in Q$ and some input symbol $x$, $O(q_i, x) \neq U(q_i, x)$.

2. **Transfer error:** any error in the transition from one state to the next. Defined formally, a transfer error implies that for some state $q_i \in Q$ and some input symbol $x$, $\delta(q_i, x) \neq t(q_i, x)$.

3. **Extra-state error:** there exists an extra state in the implementation. Defined formally, $\exists q_i \bullet q_i \in Q \land q_i \notin S$.

4. **Missing-state error:** a missing state is another type of error. If the machine representing the design is minimal and complete, a missing state implies an error in the IUT. Defined formally, $\exists q_i \bullet q_i \in S \land q_i \notin Q$.

Overall, there is a variety of methods proposed to generate test sequences for FSMs. Survey articles on test generation for FSMs include those by Wang and Hutchinson (1987), Sarikaya (1989), a very thorough survey by Lee and Yannakakis (1996), a synthetic survey presented in the form of bibliographic notes by Mathur (2008), and a major survey by Hierons et al. (2009).

One fact, which although far from being unique is still interesting to note considering the formal rigour of the area, is that a lot of work done in testing of FSMs appears to be almost duplicated by different researchers working independently. It appears that there are similar techniques labelled with different names or almost identical techniques considered for different types of graphs. For example, the idea of covering pairs of edges in the graphic representation of a FSM, is mentioned in 1976 by Pimont and Rault (1976). In 1991 Fujiwara et al. (1991) revisited the same idea and extended the pairs of edges to arbitrary lengths (n-switch); later by Offutt et al. (2003) this is called transition-pair. At an intuitive level, there seems to be scope for refining a number of what are normally labelled as different approaches, into the same method or procedure. On a deeper study of literature, it was found that the same fact has been reported by Amman et al. (2008). Even so,
to the best of my knowledge, this fact does not appear to have been analysed further. In a synopsis of the bibliography for the field, which follows in the next section, it was chosen to acknowledge the represented approaches as they are commonly referred to in literature\(^ {27}\). In future work it could be of interest for the field, to present a reduced number of approaches, which would unify the communalities of all the parallel work.

### 3.2.1. Synopsis of Approaches for Testing FSMs

Moore’s seminal paper (1956) borrowed the term “gedanken-experiments” from physics to convey the conceptual nature of drawing conclusions on the internal state of a machine from its observable behaviour. Kohavi (1978), published a book on the state of technology on testing of finite state machines. In an interesting twist of the then ‘cold war’, reportedly in the 60’s and 70’s there was a lot of research by a number of Soviet authors, which were not translated or known outside of the former Warsaw Pact countries. Well known approaches for generating tests from FSMs include, Gönenç (1970) who based his technique on the identification of distinguishing sequences; Chow’s W method (1978); and the partial W method (Wp) (Fujiwara et al. 1991), which generates shorter test sequences without apparent loss of efficacy in finding faults.

There is also a cluster of approaches based on transition tours (TT), for example Naito and Tsunoyama (1981), and Uyar and Dahbura (1986), which proposed using the Chinese postman tour algorithm for the generation of the optimal transition tour. Whereas, Sarikaya and Bochmann (1984) worked on finding upper bounds on the test sequences length for Naito and Tsunoyama’s TT method and Chow’s W method.

One very important development on FSM testing was the introduction of the concept of Unique Input Output (UIO) sequences, which are used to identify

\(^{27}\) It was felt that discussing the minutiae of which parallel work features where, would diverge the focus of this thesis.
the state reached after testing a transition and which attracted a good deal of research interest and work. UIOs in some capacity or another, are used by many test generation techniques. For example while Sabnani and Dahbura (1988) proposed the UIO sequence approach, Shen et al. (1992) proposed to optimize the UIO method, by using multiple UIO (MUIO) sequences for each state in the FSM. Subsequently, Aho et al. (1991) used the rural Chinese postman-tours to reduce the size of tests generated using the UIO method; Miller and Paul (1993) proposed an algorithm for the generation of optimal length UIO sequences under some conditions; while, Naik (1997) proposed an algorithm for the computation of minimal length UIO sequences when the later exist.

All these methods impose assumptions on the IUT, which can prove too restrictive for real systems. These limitation concerns are elucidated further in Section 3.2.5 of this thesis.

For all the propensity that FSMs have for modelling embedded systems, they do not cater easily for scenarios where there is data manipulation within a state or during transitions. Extended finite state machines (EFSMs), which in essence are FSMs augmented with variables, do better at this task.

### 3.2.2. Synopsis of Approaches for Testing EFSMs

There are a number of methods, which have been proposed for test case generation based on EFSM specifications, including (Sarikaya and Bochmann 1986), Wu and Chason (1989), Ural and Yang (1991), Wang and Liu (1993), Chanson and Zhu (1993), Huang et al (1995), Tae-Hyong et al (1998), and Uyar and Duale (2000).

One of the main issues in test generation for EFSMs is that, due to their predicates and conditional statements, test sequences generated via traditional control-flow approaches such as those used for FSMs can result in non-executable test sequences. In test generation based on EFSMs these control-flow approaches need to take in consideration data-flow approaches in order to overcome the of non-executable test sequences issue. The above methods do not consider the executability problem.

Some of the exceptions that do consider the executability problem include (Chanson and Zhu 1993), and yet their method cannot produce many executable test cases. In (Huang et al. 1995), a data-flow approach, which does not test the control-flow of the EFSM, has been complemented with a type of reachability analysis to help mitigate the executability problem.

One interesting development in terms of graphical representations of state-based behaviour was proposed by Harel (1987). Harel’s Statecharts model complex systems via diagrams using a notation that is an extension of finite state machines (EFSM). Their graphical notation, to some extent, bridges the semantic gap between the ‘heavy-weight’ formal methods and the ‘typical’ software engineer. A number of existing notations or toolsets such as Unified Modelling Language - UML (OMG), Specification and Description Language – SDL (ITU), Rational Statemate (IBM) and M/S/S (MathWorks 2008), which are already in wide industrial use, ease their adaptation.

3.2.3. Synopsis of Approaches for Testing Statecharts

In its simplest form, a Statechart is a graphical representation of an EFSM in, which each transition has an operation and a guard. The guard gives the preconditions of the transition. Each state is represented by a rectangle and transitions by arrows, which are labelled with the guard and the operation.

Many variants of Statecharts have emerged aiming to improve on Harel’s (1987) original notation or to tailor it towards a particular domain, including
(Maraninchi 1991, Jahanian and Mok 1994, Leveson 1994, Büßow et al. 1998, Galloway et al. 1998). The notation of the different variants can differ in both syntactic and semantic aspects. Syntactical differences typically include the different ways that transitions between states can be triggered (variable value driven, event driven, time driven), a presence or lack of state history etc. The different notations used in Statecharts also impose different semantics, such as using algebraic or model-based specifications, assumptions dealing with determinism, etc. In fact, a precise definition of the semantics of Statecharts has proven quite challenging and a number of authors have worked in formalising the semantics of Statecharts via an impressive number of techniques, which include (Burton 2002, Damm et al. 1997, Harel and Naamad 1996, Lüttgen et al. 1999, Uselton and Smolka 1994).

This research focuses on methods that can be used with existing toolsets, which are commonly used in industrial practice. Since in this research we are considering the development process of the embedded software, with a particular focus on automotive embedded software, the Matlab/Simulink/Stateflow environment, which is one of the most commonly used toolsets in the automotive domain is of particular interest.

The specific semantics adopted by Stateflow for its Statecharts have been in the focus of a few authors including (Tiwari et al. 2003, Scaife et al. 2004, Hamon 2005, Hamon and Rushby 2007), with Hamon and Rushby (2007) proposing a formal operational semantics for Stateflow. It is outside the scope of this thesis to contribute into the formal semantics of the Stateflow Statecharts, hence existing formalisms will be adopted. A discussion of Statecharts as they are implemented in Stateflow follows in the next heading.

In terms of test generation for Statecharts, a number of approaches focus into flattening Statecharts or converting them initially into EFSMs, LTS (Labelled Transition Systems) or FSMs. For example, Binder (1999) proposed to flatten the Statecharts and then to use a round-trip path method testing, which was an adaptation of the W method. Hong et al. (2000)
proposed an approach, which is based on deriving EFSMs from Statecharts. While Santiago et al. (2006) propose an approach, which is based into transforming Statecharts into FSMs.

Using simulation, Briand et al. (2004) analysed the cost-effectiveness of Statechart-based testing coverage criteria. Despite all these test criteria, as well as comparisons among them in terms of cost-effectiveness, for both FSM and Statecharts models, it seems that environments for automating the test case generation activity taking into account real complex software are not very common and if they do exist, are proprietary.

3.2.4. Statecharts as Implemented in Stateflow

Stateflow is one of the graphical modeling languages of the M/S/S environment. It is used to model the state-based behavior of systems and their discrete control logic\textsuperscript{28}. It uses a combination of Statecharts and flowchart diagrams. The Statecharts are based on Harel’s Statecharts, implemented with a few modifications. Stateflow is a complex language, with numerous features, which can have a non-intuitive runtime behavior. The actual definition of the language lacks any formality and its given by what Hamon and Rushby (2007) call somehow metaphorically ‘simulation semantics’. That is to say that the documentation that comes with the product (close to 1000 pages) does not give any formal operational semantics for the language; the features of the language are described informally and mainly through examples.

Clarifying what are the specific semantics that Stateflow has adopted for its Statecharts has been in the focus of a few authors including (Tiwari 2001, Scaife et al. 2004, Hamon 2005, Hamon and Rushby 2007), with Hamon and Rushby (2007) proposing a formal operational semantics for Stateflow.

Statecharts in Stateflow can have states, transitions, junctions, events, actions and conditions and they can be hierarchical. The States can include

\textsuperscript{28} While Simulink is usually used to capture the continuous behavior of a model.
entry, during, exit and on-event actions, executed respectively upon entering, during, exiting, or when the focus remains on a state after the last occurring event (i.e. no transition that leads out of that state is valid). States are defined using a somehow bespoke syntax as featured in Text Box 4.

Text Box: 4

```
State_Label /
entry: /* entry_actions */
during: /* during_actions */
ext: /* exit_actions */
on_event_name: /* on_event_actions */
```

The transitions can also have labels that define the transition’s behavior, which is given in the following syntax (Text Box 5):

Text Box: 5

```
event [condition] {condition_action} / transition_action
```

The transition label itself and all of its separate segments can be omitted.

Statecharts can also incorporate junctions, which provide decision points for transitions. There is one specific subtype of junction called a ‘history junction’, which, based on the history of the previous simulation step, can be used in ‘parent’ states to decide which of the ‘dependent children’ states will be activated next.

User defined functions can also be called from inside action labels, potentially allowing any computation to take place during the evaluation of a label.

As implied by the syntax snippets given above, in addition to the graphical language, Stateflow also uses an action language for its labels. This action language is an imperative language very similar to C.
Tiwari (2001) has given a formal definition for the Stateflow charts, which has been paraphrased below (Text Box 6):

A tuple $SFC = (V, E, S, T, f)$, where:

1. $V = V_I \cup V_O \cup V_L$ is a finite set of typed variables that is partitioned into input variables $V_I$, output variables $V_O$ and local variables $V_L$.
2. $E = E_I \cup E_O \cup E_L$ is a finite set of events that is partitioned into input events $E_I$, output events $E_O$ and local events $E_L$.
3. $S$ is a finite set of states, where each state is a tuple consisting of three types of actions: entry, exit, during. Here, an action can be either an assignment of an expression to a variable or an event broadcast.
4. $T$ is a finite set of transitions, where each transition is given as a tuple $(src, dst, e, c, ca, ta)$ in which src $\in S$ is the source state, dst $\in S$ is the destination state, $e \in E \cup \{\epsilon\}$ is an event (If $e$ is $\epsilon$, then the transition can fire on any event), $c \in WFF(V)$ is a condition given as a well formed formula in predicate logic over the variables $V$, and $ca, ta$ are sets of actions (called condition actions and transition actions, respectively).
5. $f: S \rightarrow (\{and, or\} \times 2^S)$ is a mapping from the set $S$ to the Cartesian product of $\{and, or\}$ with the power set of $S$ and satisfies the following properties: (a) there exists a unique root state $S_{root}$, i.e., $S_{root} \in \bigcup_{s \in S} descendants(s)$, where descendants($s$), is the second component of $f(s)$, (b) every nonroot state $s$ has exactly one ancestor state, that is, if $s \in descendants(s_1)$ and $s \in descendants(s_2)$, then $s_1 = s_2$, and (c) the function $f$ contains no cycles, that is, the relation $<$ on $S$ defined by $s_1 < s_2$ iff $s_1 \in descendants(s_2)$ is a strict partial order. If $f(s) = (and, \{s_1,$
Stateflow circumvents non-determinism by relying on a set of static, hierarchical priority rules which decide the order in which the objects are executed. The states are ordered based on their position in the chart; first top to bottom then left to right. Transitions with the same hierarchy level are ordered based on the content of their labels, in descending priority:

1. Labels with events and conditions;
2. Labels with events;
3. Labels with conditions;
4. No label.

If this ordering does not produce a deterministic result, the transitions are ordered by their position in the chart following a 12 o’clock rule (ordering them clockwise starting from a 12 o’clock position).

Stateflow provides all the simulation and code generation functionalities as the other languages of the M/S/S environment. It would benefit from better documentation where the complex functionality of the language is described in a precise and unequivocal manner. In this regard, an ‘agreed upon’ formal definition of the full language would be beneficial and would also facilitate the connection of Stateflow with formal tools. Increasing the level of formal development when and as often required by the software quality standards, which regulate the relevant industrial sector.

3.2.5. Reflections on the Limitations of the Reviewed Techniques

There is an impressive body of work targeted at formal approaches for testing state-based systems. Some of the main algorithms at the core of testing FSMs, EFSMs and Statecharts are concerned with finding what is the current state of the IUT (for example in the W and the Wp methods and in other methods based on characterizing or distinguishing sentences), or, checking whether the IUT happens to be in a specific state (for example in the algorithms using UIO sequences).

While most of the algorithms do present elegant mathematical solutions, they also pose requirements, which can be too restrictive for the IUT.
Typical requirements include the IUT having to be minimal, complete and fully connected. They can also require that each test sequence has to start at a specific initial state. For example the W and Wp methods, assume a reset feature. In other words, they require that after each test sequence, the system can be accurately reset to the initial state.

Resetting the system after each test sequence can hide a plethora of software errors such as say additional state errors, which only appear after long input test sequences. Moreover, an assumed reset feature does not cater well to test hybrid systems, which can be triggered by a combination of data and events.

A common assumption of these techniques is that errors are not masked by subsequent errors in the checking sequence.

The techniques for testing EFSMs have the additional problem of not knowing how many of the generated tests will be executable. Most of these techniques find it difficult to combine data-flow with control-flow in their algorithms.

When testing Statecharts, the interaction between data and the flow of control and data manipulation during transitions and within states translates in the fact that a number of errors are not necessarily detected by a single execution of a transition or state. The same data-control interaction also complicates the task of accurately driving the Statechart to a specific transition.

Statecharts also often have hierarchies, which can have orthogonal components. Most of formal testing methods for Statecharts start by flattening its hierarchy via multistep techniques. Testing the communication of orthogonal components and flattening hierarchies add another layer of complexity, and often translate in a reduced automation for the test generation and/or added restrictions for the IUT.
To the best of my knowledge there is no work that can automatically generate test sequences to simultaneously test both the transfer and output functions of Harel’s Statecharts. It appears that approaches to automatically generate tests based on the qualities of Statecharts for real complex software are not common. Overall, all the approaches share the problem of state-explosion. Furthermore, a considerable number of researchers compare their proposed techniques to other idealized versions of techniques, making the uptake by industry for use in real complex systems quite challenging.

This brings the focus to one of the triggers behind this research, more specifically, to test generation techniques which can incorporate formality and are informed by what is currently usable or prone to be used in industry.

3.3. Conclusions

The latest QA standards require specific testing metrics such as MC/DC coverage for safety-related software. The standards also require the incorporation of formality in the testing process. Despite the evidence on the benefits of formal methods, their uptake by industry remains unsatisfactory. Graphical formal notations tend to fare a bit better. Statecharts, which in essence are a graphical representation of EFSMs, are relatively well supported by tools. Yet, their hierarchical structure and data interaction can be very challenging for automatic test generation.

Overall, while there are an impressive number of elegant formal testing approaches that aim to generate tests automatically, they tend to be too restrictive and difficult to be used for complex industrial systems.
PART 2
Research Contributions and Methodology
4. Research Contributions

“At this moment, the laurel bush, which had hitherto not spoken, said "Psst!"

P.G. Wodehouse, “Summer Lightning”, 1929

This chapter presents the contributions of this research. It presents a new testing method called CTM\textsubscript{CONTROL} and describes a Formalization of a Reactive Testing Process for testing automotive safety-related software in a model-based approach. The Formalization of a Reactive Testing Process also includes a taxonomy for test reactivity and proposes changes in a previously published taxonomy for model-based testing. The methodologies and the design of this research are also discussed.

4.1. Overview of the Contributions

Model-based testing often relies on specification-based behaviour models of the system. One well-known approach is to capture the behaviour of the SUT via state-based machines and charts. This research is focused on the SUT modelled via Statecharts as they are implemented in the Stateflow component of the M/S/S environment. This, since one of the aims of this research is to use toolsets, which are already in common use in the domain of interest and around 50 % of functional behaviour for embedded systems is modelled in the M/S/S environment (Helmerich et al. 2005).

As discussed in Chapter 3 of this thesis, while there is considerable research in generating tests for Statecharts, most of the proposed solutions have the common limitation of being based on a number of assumptions and preconditions, which can prove too restrictive or even unrealistic for complex industrial systems.
This thesis proposes a new testing method called $\text{CTM}_{\text{CONTROL}}$, which complements a popular testing method with MC/DC coverage capabilities as per one of the objectives posed by the latest QA standards (ISO 2011). Additionally this thesis proposes the Formalization of a Reactive Testing Process (FRTP) which builds a pathway between reactive testing and formal testing. The FRTP aims to help the incorporation of formality in the automotive software testing process. The FRTP includes a new Taxonomy of Reactive Testing (TRT). Also, based on the focus on reactive testing, changes are proposed to a model-based testing taxonomy, which was initially proposed by (Utting et al. 2006) and later enhanced by (Zander-Nowicka 2009). The proposed new method ($\text{CTM}_{\text{CONTROL}}$), the FRTP and the TRT target the field of safety-related automotive software.

For clarity of explanation, the chapter first discusses $\text{CTM}_{\text{CONTROL}}$. After that the Taxonomy of Reactive Testing is presented on its own. The discussion continues with the presentation of FRTP. The discussion of FRTP includes examples of all the dimensions of the TRT formalized in Z. The examples are meant to be used as information models by automotive testers who need to incorporate formality in their testing process. The last section of the chapter discusses the methodologies chosen for this research.

### 4.2. $\text{CTM}_{\text{CONTROL}}$

The most difficult aspect in software testing is to choose the right tests. Coverage is one of the important metrics used to evaluate the quality of a testing process. Coverage, as discussed in Section 2.7 of this thesis, is a multifaceted concept, which among other things, translates into the issue of what type of coverage should and can be chosen. Furthermore, coverage alone is an insufficient criterion to measure the quality of tests.

Coverage metrics combined with functional testing are recommended by literature and standards such as DO-178C and ISO 26262. ISO 26262,
which covers automotive software recommends MC/DC coverage to be used in addition to functional tests.

The Classification Tree Method (CTM), as described in Section 2.3.5 of this thesis, is a model-based testing approach, which is widely used for functional testing and is well supported by the tool-chain in the automotive industry (Conrad et al. 1999). In essence, CTM consists of partitioning the input domain of the SUT into equivalence classes. The classes correspond to ranges of values over the different input parameters, within which the SUT is expected to behave uniformly. The test cases are generated by combining the values selected from the different partitions.

CTM, while providing a clear strategy for systematic testing, gives the engineer no control over the degree of model coverage, and can leave parts of the control logic of the model unexercised. CTM, in this, behaves in line with the typical requirement-based testing approach and does not provide assurances regarding the presence or lack of unintended functionality in the SUT. Since unintended functionality in a system is a potential source of unsafe behavior, complementary approaches are required.

In avionics software (for example in the DO-178B (RTCA 1992) standard and its updated version DO-178C (RTCA 2011)), this issue is addressed by introducing the concept of structural coverage analysis. Here, the purpose of structural coverage analysis is to “determine which code structure was not exercised by the requirements-based test procedures” (RTCA 2011). This concept of structural coverage analysis is not to be confused with structural testing, the latter understood as a set of activities that exercise the software with tests generated based on the source code, not the requirements.

What is proposed in this thesis is to enhance the CTM method with a control aspect. The control aspect in the proposed CTM\textsubscript{CONTROL} focuses on the logical expressions guarding the transitions of the system represented via Statecharts\textsuperscript{29} and provides MC/DC coverage for them. CTM\textsubscript{CONTROL}

\textsuperscript{29} Here we are considering systems modelled via Statecharts as they are implemented in Stateflow. That said, the approach in principle is equally viable for systems captured via labelled systems or any form of “enriched” state machines.
allows to test the dependencies among the input values and the control logic of the SUT.

In the proposed CTM\textsubscript{CONTROL} approach, in addition to the tests generated as per the classic CTM, tests are generated so as all atomic predicates in a state-transition are set to \textit{true} with the output observed to be \textit{true} and each and every input is set exclusively to \textit{false} with the output observed to be \textit{false}.

The underlying (scientific) hypothesis of the proposed new method is that CTM\textsubscript{CONTROL}, can detect a group of errors, which would not be normally caught via the classic CTM.

To clarify this idea, I shall consider system \( S \) (Fig. 16).

![Figure 16: System S](image)

For ease of explanation, I shall focus only on the transition from state ‘2’ to state ‘3’, labelled as ‘\( L \)’. As it is often the case in control systems, the state-transition \( L \) is guarded by a combination of logical predicates (in this case three predicates) denoted for ease of explanation as \( A, B, C \):

\[
L = (A \wedge B) \lor (\neg C) \quad (1)
\]

If \( L \) is tested via the traditional CTM method, the tests are generated by designing equivalence classes for the input parameters of \( L \) and choosing one value from typically a range of possible values for each of the partitions. This does not guarantee that the test process will achieve any specific model coverage. What is more, a test process, which follows the traditional CTM method by focusing on parametrizing the equivalent
partitions of the input parameters would not directly target the ‘¬C’ portion of the transition guard. CTM would also not target each and every combination of the atomic logical predicates exclusively evaluating to *true* and *false* and the effect they have on the SUT. How does the SUT behave when only one of the predicates is true? How does the SUT behave when each of the atomic predicates exclusively evaluate to true? How does the SUT behave when \( L \) evaluates to false? The above are but a few of the questions, which are left unanswered by the classic CTM.

Predicates such as ‘¬C’ on its own or the effect that each of the separate atomic logical predicates evaluating to *true* and/or *false* has on the SUT are important to test not only for the presence of the required functionality, but crucially, to test for the absence of unwanted functionality.

In other words, since CTM does not necessarily target these combinations of logical values it can be deduced that it does not uncover some types of control errors that could be present in a system. This includes some types of errors that can lead to unwanted activations of parts of the system or unwanted feature interactions. As for example it was the case of a faulty airplane altimeter causing the crash of the THY Boeing 737-800 on its approach to Amsterdam in 2009. On that occasion, an erroneous altimeter reading caused the autopilot to deduce that the plane was landing and as a consequence to automatically reduce the fuel intake, causing loss of velocity while in high altitude and crash causing loss of life.

In keeping with the classification tree concept, which underpins the classic method, CTM\textsubscript{CONTROL} translates the MC/DC aspect into a tree format. The tests for the MC/DC aspect are generated via a classification tree where the different classifications correspond to the logical predicates that make up the state-transition guards. Each of these classifications is then further partitioned into *true* and *false* values and the tests are generated by toggling each condition between *true* and *false* and toggling each decision between *true* and *false*. More specifically, in the proposed CTM\textsubscript{CONTROL} approach, in addition to the tests generated as per the classic CTM, tests are generated such as all atomic predicates are set to *true* with the output observed to be
true and each and every input is set exclusively false with the output observed to be false.

Figure 17: Test cases generated for the control aspect of CTM\textsubscript{CONTROL}.

In the example of Figure 17, the additional\textsuperscript{30} test cases generated to incorporate MC/DC coverage for $L$ are:

<table>
<thead>
<tr>
<th>Test No.</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$\neg$C</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

It is important to highlight the fact that here are covered only the MC/DC tests which can be generated. In some cases, for instance, one of the atomic predicates can be dependent on another and MC/DC coverage is a

\textsuperscript{30} In addition to the tests generated via the classic CTM method.
mathematical impossibility (without changing the logical structure of the SUT).

Figure 18 depicts an abstract-level mapping of CTM\text{CONTROL} against the backdrop of specification-based testing and control-flow testing.

As shown in Figure 18, CTM\text{CONTROL} can be used at different test colour levels, depending on the visibility that the tester has on the internal structure of the SUT.

It is common for automotive embedded software to be tested by a third party. For example the developer without disclosing the details of the implementation can get a product certified based on certification tests performed by a third party (such as a certification agency). Certification tests are specification-based and, at that level, the MC/DC aspect of CTM\text{CONTROL} can be used to test those logical decisions and conditions that are visible at the certification specification level. Having a formal specification can facilitate this task.

CTM\text{CONTROL} can also be targeted at a finer granularity for functions, which are part of the SUT and need to be developed according to the requirements
of the higher ASIL requirements. Here it is important to highlight the fact that even at this off-white- to- white-box level approach, testing can be designed independently from the implementation. In other words testing can still be based on the parameterized partitions of the SUT input parameters’ values and the control logic within the function. The light-colour-box visibility here refers to the visibility of the control structure within the code rather than to the visibility of the code in its entirety. This is the type of control structure, which is based on the specification and known from very early in the development process.

Summarizing, CTMCONTROL has been developed by extending an existing method so as not to introduce a steep learning curve for automotive software testers. When compared to the method, which it improves upon, CTMCONTROL can detect an additional group of errors. The approach, in addition to the benefits of the classic CTM, also fulfills the MC/DC objective posed by the QA standards. As it is discussed in the next chapter, CTMCONTROL has been prototypically realized in the M/S/S environment, which is popular in the automotive industry.
4.3. The Taxonomy of Reactive Testing

In this research, reactive testing is defined as testing where the next test to be executed depends on the newly observed behaviour of the system under test. Output values of the SUT, temporal relations or constraints among parameters, the success or failure of some component of a test sequence, are all examples of behaviour that needs to be catered for by reactive testing.

The relevance of a specific type of test reactivity to be included in a testing process can be domain and application specific. For instance, automotive embedded software needs to satisfy hard-timing requirements while operating within the restrictions imposed by sensors, actuators, the physical position of the Electronic Control Unit in which the software will be deployed, in-vehicle network protocols, gateways etc. A reactive testing process needs to react to observed behaviour such as “if after signal x and after signal y, parameter A’s value is over a certain threshold, then run a specific test which observes the value of parameter A when the ABS (Anti-lock Brakes System) is engaged” or to cater for scenarios where a precise sequence of events and actions need to happen within a precise period of time in the range of a few μsec.

But, what are the specific attributes of an application that determine what kind of test reactivity is relevant? Would it be beneficial to elicit the different types of reactivity that different companies and testers are looking for in their testing? Would a better understanding of the commonalities and differences of test reactivity relevant for different domains, help to design guidelines, which help the decision making of the tester and help tailor generic reactivity guidelines into an application-specific situation? The above are but a few of the questions that have triggered this research’s interest into test reactivity.

As a first step towards answering these questions, this research focuses on proposing an initial skeleton of a Taxonomy of Test Reactivity (TRT). The test reactivity types are organized in a taxonomy with a view that this
elucidation and ordering of information could be a starting point into identifying relationships, patterns and factors that influence the relevance of different reactivity dimensions for a specific testing process. A tester using the proposed usability model can choose from the presented dimensions, modify them or add new ones.

The taxonomy of test reactivity, which is proposed here, was built in a bottom-up approach. The work started with identifying all the different types of test reactivity which are reported to be of interest while testing embedded automotive software. This task was facilitated by the fact that there are a number of researchers that have focused on reactive testing, and an especially comprehensive description of the types of test reactivity can be found in (Zander-Nowicka 2009). After the identification of the types of reactive testing, they were classified in dimensions via an iterative approach. In each iteration the classifications were refined using a combination of walk-throughs and usability testing methods. The refinement process was driven by three iteratively posed questions:

1. Does all the elucidated content fit (i.e. could all the types of test reactivity be classified via the taxonomy)?
2. Can any of the classifications be merged with each other?
3. Can any of the hierarchical relationships be flattened?

This process identified the taxonomy of test reactivity as depicted in Figure 19. In the taxonomy, test reactivity is organized into three “super” dimensions which contain a number of “sub” dimensions. To lessen the semantic confusion, all the different classes of behaviour to which the testing process can react to, independently from which taxonomic level they fit into (i.e. independently on whether they are a “super” or a “sub” dimension), are referred to as a reactivity dimension.

In the following sections the different dimensions of test reactivity are discussed and possible instantiations (examples) are given for each dimension.
TRT 1 Coverage Criteria Related Reactivity Dimensions

The reactivity dimensions categorized as *coverage criteria related* include dimensions such as:

TRT 1.1 Coverage of states and state-transitions.

This dimension covers scenarios where the testing process needs to react to the coverage of states and state-transitions of the SUT model. For example, let us assume that one of the testing objectives stipulates that all of the state-transitions of the SUT have to be covered. First of all this means that test generation needs to include an algorithm that addresses state-transition coverage. Second, when test prioritization is required, if a specified
coverage of state-transitions has not been achieved within a specified time interval, then the test harness should react and execute a test sequence that covers all the state-transitions of the SUT.

TRT 1.2 Coverage of conditions and decision points.

This dimension covers scenarios where the testing process needs to react to the coverage of the condition/decision points. For example, let us assume that one of the testing objectives stipulates that over a specific percentage of the decision points need to be covered. If this is not achieved over a specified time interval, then the test harness reacts by executing a test sequence that covers all the decision points.

TRT 1.3 Coverage of testing objectives.

Similarly with the two previous dimensions, the testing process will prioritise the coverage of testing objectives over all the other possible test selection scenarios.

TRT 1.4 Coverage of specification.

The testing process reacts to the specification coverage. For example, if the test process has been running for a specific amount of time and specification coverage is still under a threshold, in a time-restricted scenario, subsequent testing will focus in ensuring the required specification-coverage.
TRT 2 Input / Output and Temporal Space Related Reactivity Dimensions.

The reactivity dimensions categorized as *input/output and temporal space related* include dimensions such as:

TRT 2.1 Test result: Pass/Fail.

Here the test harness reacts to the failure or success of a specific test. For example if a specific test fails (let us call it test number \( x \)), the test harness reacts by executing a test case or a test sequence which has been predesigned to follow the eventuality of test number \( x \) failing.

TRT 2.2 Parameter values.

The testing process reacts to a parameter having a specific value (including zero and other boundary values). For example if during the testing process the vehicle’s battery voltage is observed to fall under a specified threshold, the test harness reacts by turning on the climate control (a variant of load testing).

TRT 2.3 Parameter relationships.

The testing process reacts to the detection of a combination of values for specific parameters. For example, if during the testing process the lateral acceleration of the vehicle is observed to be over a specific threshold of 9 m/s\(^2\) while the velocity of the vehicle is over a threshold of 28m/s, the test harness reacts by turning on the Automatic Cruise Control system.

TRT 2.4 Signal properties.

The testing process reacts to the detection of a specific property in a signal. Here the term *signal property* includes aspects such as the signal remaining constant over a sampling period, a maximum or minimum value of a signal, of periodicity in a signal, of the signal ‘disappearing’ after or before a
specified time $t$, etc. In other words characteristics of signals, which are typically studied by signal processing disciplines.

TRT 2.5 Temporal testing.

The testing process reacts to timing issues such as the time it takes to a certain test sequence to execute, how long the system has been running, etc.

TRT 2.6 Scenario patterns.

The testing process reacts to some predefined patterns of combined timing and parameter issues such as scenarios of type: if after $t$ seconds signal $s$ is constant AND after $t_1$ seconds parameter $p$ (measured by sensor A) evaluates higher then parameter 2 (measured by sensor B), then run the predesigned test $y$.

TRT 2.7 Triggers.

The testing process reacts to the detection of a specific event which under specific conditions should trigger a specific SUT behaviour. The test harness reacts by executing a test sequence, which examines whether the correct SUT’ behaviour was triggered.

TRT 3 Knowledge Base Related Reactivity Dimensions

The reactivity dimensions categorized as knowledge base related include dimensions such as:

TRT 3.1 Black spot based.

The testing process reacts to the detection of some predefined black spot scenario, which in previous versions of the SUT has proved to cause error-prone behaviour. For example during the testing process of an electronic stability control system, the vehicle is driving on ice (the friction coefficient
with the road ($\mu$ is very low) and the gyroscopic sensor in the car detects that the car’s direction does not agree with the driver’s intent (as measured by the steering wheel sensor). The testing process reacts by exercising the behaviour of the car while the driver attempts an emergency brake in these conditions.

TRT 3.2 Rare but safety important.

The testing process reacts to the detection of some predefined scenario which might occur rarely but if it occurs and the SUT’s behaves erroneously, it can have serious safety repercussions.

Overall, the proposed taxonomy is meant to be used as an evolving artefact, where the tester can choose from the depicted reactivity dimensions or add new ones. The TRT proposed here is also part of the Formalization of a Reactive Testing Process (FRTP), which is discussed in the following section.
4.4. Formalization of a Reactive Testing Process

QA standards such as ISO26262 (ISO 2011) ask for formal processes to be used in the development of critical software. Hence, it was deemed important to illustrate how the proposed CTM\textsubscript{CONTROL} can be implemented within a process, which incorporates formality. Considering the different obstacles in the way of formality’s uptake by industry, this thesis proposes the Formalization of a Reactive Testing Process, which breaks down the testing process into relatively simple and manageable steps. The FRTP defines ordered steps and a flow of logic on how to move between them, which serve as methodological pointers to help with the integration of formality in the testing process. Examples are provided for each step, aiming to counteract some of the perceived complexity.

In addition to the benefits of the rigorous semantic and powerful verification capability of formal and mathematical techniques in software engineering, of an equal importance is their benefit of clear notions and terminology (Broy 2009). To paraphrase Broy (2009) the later benefits are equally important to help understanding between engineers and reducing waste of time in “confusing discussions”.

Informed by this, the Formalization of a Reactive Testing Process uses Z notation as an intermediary notation to formalise aspects of the testing process as per the proposed usability model.

The reasons behind the choice of Z notation include its convenience to represent Statecharts and their transitions, the fact that it is relatively well supported by verification tools; it can be used to verify properties; and potentially generate test sequences. Z notation has also been previously successfully used in combination with Statecharts by a number of researchers (Burton 2002, Büssow 2003).
Since the focus of this thesis is on standard-compliant testing, the formalization is done at the specification level. This decision was taken due to the fact that most (if not all) automotive software needs to go through some third party acceptance testing and certification before passing through the supply chain. Typically the acceptance and certification activities, which underpin the communication among the different stakeholders, pertaining software that has to comply with a specific standard (such as ISO 26262) are done at a specification level. Furthermore, the presence of a formal specification helps the tester to be clearer on exactly what it means for a system to pass a test (Hierons et al. 2009).

While formalization permeates the entire usability model, the degree to which the more formally-heavy Z notation is used is tailorable. That is to say that depending on a specific project’s requirements, timing and financial constraints, the tester can choose in which steps to omit the use of formal specification via Z.

As it can be seen in more detail in Section 5.2.3 and APPENDIX B of this thesis, Z is used alongside the M/S/S environment in an automotive case study in order to show how a formal testing process can be split in small and concrete steps all the while keeping the implementation in an environment, which is familiar for industry.

CTM\textsubscript{CONTROL} and the FRTP can be used independently from each other. That said, in this thesis CTM\textsubscript{CONTROL} is implemented and executed in the M/S/S environment and is also illustrated within the FRTP\textsuperscript{31}. Conceptually, the M/S/S environment is used to illustrate some of the concrete aspects of CTM\textsubscript{CONTROL} and the FRTP is used to illustrate some of the methodology-related abstract aspects of CTM\textsubscript{CONTROL} (Fig. 20).

\textsuperscript{31} Please see APPENDIX B in this thesis.
The steps of the FRTP are depicted in Figure 21 and are discussed in the following sections.
The ideas behind the usability model are illustrated in Z notation. The examples are chosen as simple as possible for clarity of the argument and with a view of avoiding the proverbial fear of adoption from industry.

4.4.1. Step 1: Formalization of Test Objectives

The process starts with formally stipulating the *test objectives*\(^{32}\) according to the quality assurance requirements for the project at hand.

The test objectives can include (but are not limited to) issues such as:

- Demands imposed by the relevant guidelines and/or certification standards (such as for example the recommendations for the required SIL, ASIL, etc).
  The abstract aspects of these demands are translated into concrete and implementable actions. For example, when developing software functions so as to conform to SIL 4, in addition the other requirements, this objective will translate into the demand that the software is developed and tested following a formal process.
- Requirements on the length of test sequences. This due to the fact that, especially in the presence of hardware, some faults are only discovered after the system has been running for a certain time.
- Requirements to cover structural features of the SUT such as programming logic branches, condition/decision points, state-transitions, states, etc.
- Requirements related to the coverage of software specification.

While generally speaking, the formal notation used to capture objectives could be chosen to suit the domain of interest or the software development

\(^{32}\) The meaning of the term *test objectives* in this research is akin to *test quality metrics* in the sense that it defines the benchmark/s that the testing process has to be compared to. Since safety relevant functions in software in some occasions have to conform to quality measures that can only be measured or reasoned about qualitatively (like for example is the case for SILs and ASILs), it was felt that the term *objectives* rather than the quantitative connotation of the word *metric*, conveys the concept better.
process of the company, as already stated, this research uses Z notation, combined with the ‘formally lighter’ graphical notation of Statecharts.

For instance, the formal capture in Z notation of a test objective to achieve a minimum of 90% of model coverage (seen as a coverage benchmark) is illustrated\(^{33}\) below (preceded by an informal description in natural language).

The following Z schema specifies that testing will be deemed incomplete \((\text{TestingComplete}=\text{F})\) until the test process has achieved a model coverage \((\text{ModelCoverage})\) which is equal or higher than a specific benchmark \((\text{CoverageBenchmark})\).

\[
\begin{align*}
\text{ModelCoverageObjective} \\
\text{ModelCoverage, CoverageBenchmark: Z} \\
\text{TestingComplete: BOOLEAN} \\
\text{ModelCoverage < CoverageBenchmark} \Rightarrow \text{TestingComplete = F}
\end{align*}
\]

Using a formal notation such as Z, lays the groundwork for verification of critical properties. That is to say that if needed, the relevant testing objectives can be specified as lemmas and completeness, disjointness and satisfiability properties of these heuristics or correctness of the implementation can be proven via pre-existing toolsets; overall laying the groundwork for the verification of critical properties\(^{34}\).

---

\(^{33}\) The Z notation illustrations are treated as they were snippets of a full specification document. Hence, some of the definitions for variables etc that normally would accompany the schemas are omitted for the sake of brevity. The Z notation used in this section follows the ISO/IEC 13568 standard ISO/IEC 13568: Information Technology; Z Formal Specification Notation; Syntax, Type System and Semantics (1st edition) 2002, ISO/IEC.

To facilitate type-checking all the parameters are declared to be integers. As a consequence, all the numerical values within the Z schemata have been rounded up/down to their nearest integer value.

\(^{34}\) Outside the scope of this research but possible venue for future work
Rationale Behind Step 1:

a. The added clarity and understanding provided by the rigorous semantics and formal approach.

b. Specifying the test objectives in formal notation, facilitates the process of developing software according to a higher SIL and ASIL rating as per the requirements of the relevant industry standards.

c. Use of the automation potential offered by Z specifications (in some cases tests can be generated automatically or semi-automatically directly from Z schemata by techniques proposed by other researchers such as (Burton 2002)).

d. Once in formal notation, properties of interest such as disjointness, completeness etc, if needed, can be proven via general purpose theorem provers.

4.4.2. Step 2: Formalization of the Test Reactivity

In this step, the test designer chooses which specific reactivity dimensions from the TRT to include in the testing process and the reactive behaviour of interest is specified formally.

With a view to aid the use of formalization in industry, below are given examples in Z notation for each of the dimensions of the TRT. The examples are meant to serve as informational models, which can guide the ‘typical automotive software tester’ in the process of formalizing test reactivity.

4.4.2.1. Formalization examples of coverage criteria related reactivity dimensions:

4.4.2.1.1. Formalization example of the coverage of states and state-transitions reactivity dimension.

If the test sequences have been running for a time (ProgressCheckTime) longer than a specified time interval (CheckTime), the test harness should check the number of transitions covered until then (NrTransCovered). If that
number is smaller than the total number of transitions \((NrTrans)\), the next test sequence \((TestSeqCurrent)\) chosen by the test harness should be one that covers all the transitions \((AllTransitionsAlgorithm)\). Additionally the reactivity flag \((ReactFlag)\), which lets the test harness to record the fact that a reactive behaviour was captured, is set to \textit{true}.

Here \([\text{TESTSEQUENCE}]\) is the set of all possible test sequences.

\[
\begin{align*}
\text{OBJ TransitionCoverage} \\
\text{AllTransitionsAlgorithm}: \text{TESTSEQUENCE} \\
\text{TestSeqCurrent}: \text{TESTSEQUENCE} \\
\text{NrTransCovered, NrTrans}: \mathbb{N} \\
\text{ProgressCheckTime, CheckTime}: \mathbb{N} \\
\text{ReactFlag}: \text{BOOLEAN} \\
\end{align*}
\]

\[
(\text{ProgressCheckTime} > \text{CheckTime} \land \text{NrTransCovered} < \text{NrTrans}) \Rightarrow \quad (\text{TestSeqCurrent} = \text{AllTransitionsAlgorithm} \land \text{ReactFlag} = \text{T})
\]

4.4.2.1.2. Formalization example of the \textit{conditions and decision points} reactivity dimension.

If the test sequences have been running for a time \((ProgressCheckTime)\) longer than a specified time interval \((CheckTime)\), the test harness should check the number of programming decision points covered so far \((NrDecisionsCovered)\). If that number is smaller than a specific number of decisions \((NrDecisionsCovBenchmark)\), the next test sequence \((TestSeqCurrent)\) chosen by the test harness should be one that covers all the programming decisions \((AllDecisionsAlgorithm)\). Additionally the reactivity flag \((ReactFlag)\), which lets the harness to record the fact that a reactive behaviour was captured, is set to \textit{true}.

\[
\begin{align*}
\text{OBJ DecisionCoverage} \\
\text{NrDecisionsCovered, NrDecisionsCovBenchmark}: \mathbb{N} \\
\text{ProgressCheckTime, CheckTime}: \mathbb{N} \\
\text{TestSeqCurrent, AllDecisionsAlgorithm}: \text{TESTSEQUENCE} \\
\text{ReactFlag}: \text{BOOLEAN} \\
\end{align*}
\]

\[
(\text{ProgressCheckTime} > \text{CheckTime} \land \text{NrDecisionsCovered} < \text{NrDecisionsCovBenchmark}) \Rightarrow \quad (\text{TestSeqCurrent} = \text{AllDecisionsAlgorithm} \land \text{ReactFlag} = \text{T})
\]
4.4.2.1.3. Formalization example of the coverage of testing objectives reactivity dimension.
If the test sequences have been running for a time (ProgressCheckTime) longer than a specified time interval (CheckTime), the test harness should check the number of testing objectives covered so far (NrObjCovered). If that number is under a specific threshold (NrObjCovBenchmark), the next test sequence (TestSeqCurrent) chosen by the test harness should be one that covers all testing objectives (TestObjSequence). Additionally the reactivity flag (ReactFlag), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\begin{align*}
OBJ_{\text{CoverageOfObjectives}}
\end{align*}
\]

\[
\text{NrObjCovered}, \text{NrObjCovBenchmark} : \mathbb{N} \\
\text{CheckTime, ProgressCheckTime} : \mathbb{N} \\
\text{TestSeqCurrent, TestObjSequence} : \text{TESTSEQUENCE} \\
\text{ReactFlag} : \text{BOOLEAN}
\]

\[
\begin{align*}
(\text{ProgressCheckTime} > \text{CheckTime} \land \text{NrObjCovered} < \text{NrObjCovBenchmark}) \Rightarrow \left(\text{TestSeqCurrent} = \text{TestObjSequence} \land \text{ReactFlag} = T\right)
\end{align*}
\]

4.4.2.1.4. Formalization example of the coverage of specification reactivity dimension:
If the test sequences have been running for a time (ProgressCheckTime) longer than a specified amount of time (CheckTime), the test harness should check the number of specification requirements covered so far (NrSpecReqCovered). If that number is under a specific threshold (NrSpecCovBenchmark), the next test sequence (TestSeqCurrent) chosen by the test harness should be one that covers all the specification requirements (SpecReqSeq). Additionally the reactivity flag (ReactFlag), which lets the test harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\begin{align*}
OBJ_{\text{CoverageOfSpecification}}
\end{align*}
\]

\[
\text{NrSpecReqCovered}, \text{NrSpecCoveredBenchmark} : \mathbb{N} \\
\text{ProgressCheckTime, CheckTime} : \mathbb{N} \\
\text{TestSeqCurrent, SpecReqSeq} : \text{TESTSEQUENCE} \\
\text{ReactFlag} : \text{BOOLEAN}
\]

\[
\begin{align*}
(\text{ProgressCheckTime} > \text{CheckTime} \land \text{NrSpecReqCovered} < \text{NrSpecCoveredBenchmark}) \Rightarrow \left(\text{TestSeqCurrent} = \text{SpecReqSeq} \land \text{ReactFlag} = T\right)
\end{align*}
\]
4.4.2.2. **Formalization examples of the reactivity dimensions categorised as test input/output and temporal space related:**

4.4.2.2.1. **Formalization example of the test result: pass /fail reactivity dimension:**

If a specific test fails (i.e. the Boolean called TestResult is false), the next test sequence (TestSeqCurrent) chosen by the test harness should be a test sequence (TestSeqNrXX) which has been designed to follow this test sequence's failure. Additionally the reactivity flag (ReactFlag), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\begin{align*}
\text{TestFailScenario} \\
\text{TestResult, ReactFlag: BOOLEAN} \\
\text{TestSeqCurrent, TestSeqNrXX: TESTSEQUENCE} \\
\text{TestResult = F \Rightarrow} \\
(\text{TestSeqCurrent}= \text{TestSeqNrXX} \wedge \text{ReactFlag}= T)
\end{align*}
\]

4.4.2.2.2. **Formalization example of the parameter values reactivity dimension:**

If during the test process of an Adaptive Cruise Control, the value of the battery voltage (BateryVoltage) falls under 2 volts, the next test signal should force the current state of the test harness (ACC_CurrentState) to be a cruise by velocity state (State_CruiseByVelocity) where the Adaptive Cruise Control, in the absence of an obstacle, travels at a velocity preset by the driver. Additionally the next test signal should simulate an obstacle to appear in front of the vehicle (TargetPresent=T) that is controlled by the Adaptive Cruise Control. The reactivity flag (ReactFlag), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

Here, STATUS is a free type which contains all the states of a SUT modelled as a Statechart.
4.4.2.2.3. Formalization example of the parameter relationships reactivity dimension:
If during the test process of an Adaptive Cruise Control, the velocity of the vehicle controlled by the Adaptive Cruise Control \(v\) is over or equal to 28 m/s while the lateral acceleration \(a_{Lat}\) is at least 9 m/s\(^2\), the next test signal should force the current state of the test harness \((ACC\_CurrentState)\) to be a cruise by distance state \(\text{State\_CruiseByDistance}\) where the Adaptive Cruise Control keeps the car it controls at a safe distance from the obstacle in front. Additionally the reactivity flag \((ReactFlag)\), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\begin{align*}
\text{TestParameterRelationshipScenario} \\
&v, a_{Lat}: \mathbb{Z} \\
&ACC\_CurrentState: \text{STATE} \\
&ReactFlag: \text{BOOLEAN} \\
&(\forall v \geq 28 \land a_{Lat} \geq 9) \Rightarrow \\
&(ACC\_CurrentState = \text{State\_CruiseByDistance} \land \text{ReactFlag} = T)
\end{align*}
\]

4.4.2.2.4. Formalization example of the signal properties reactivity dimension:
Here \(accFnc\) is a function which models the shape of the car’s acceleration signal. If the test harness detects a maximum in the acceleration signal \((acc)\), it increases the velocity of the car \(v\) by 10 times a predesigned amount \((Step)\). Additionally the reactivity flag \((ReactFlag)\), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\begin{align*}
\text{TestParameterValueScenario} \\
&\text{BatteryVoltage}: \mathbb{Z} \\
&ACC\_CurrentState, \text{State\_CruiseByVelocity}: \text{STATUS} \\
&\text{TargetPresent}, \text{ReactFlag}: \text{BOOLEAN} \\
&\text{BatteryVoltage} < 2 \Rightarrow \\
&(ACC\_CurrentState = \text{State\_CruiseByVelocity} \land \text{TargetPresent} = T \land \text{ReactFlag} = T)
\end{align*}
\]
4.4.2.2.5. Formalization example of the temporal testing reactivity dimension:
Here the test harness reacts to the fact that a specific test sequence needs to execute in under 500 seconds. Hence if it observes that the test sequence has been running for longer \( t > 500 \), the test harness marks that test sequence as a failed one \( \text{TestResult}=F \). Additionally the reactivity flag \( \text{ReactFlag} \), which lets the harness to record the fact that a reactive behaviour was captured, is set to \text{true}.

\[
\text{TimeSensitiveTestResult}
\begin{align*}
t &: \mathbb{N} \\
\text{TestResult}, \text{ReactFlag} &: \text{BOOLEAN} \\
t > 500 &\Rightarrow \\
(\text{TestResult}= F \land \text{ReactFlag} = T)
\end{align*}
\]

4.4.2.2.6. Formalization example of the scenario patterns reactivity dimension:
Here \( \text{sigFnc} \) is a function which models the shape of a specific parameter’s signal. Here the test harness detects that after the test process was running for 100 seconds \( t = 100 \) the signal of a specific parameter \( \text{sigFnc} \) was constant for three simulated seconds. Additionally after the test process was running for another 20 seconds \( t > 120 \) parameter \( x \) reads higher than parameter \( y \). The next test sequence to be executed \( \text{TestSeqCurrent} \) will be \text{TestSeqNrAAA}. The reactivity flag \( \text{ReactFlag} \),
which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\text{sigFnc: } \mathbb{Z} \times \mathbb{N} \to \mathbb{Z} \\
m: \mathbb{N} \\
n: \mathbb{Z}
\]

\[
\forall x: \mathbb{Z} \bullet m < 10 \land ((x, m) \in \text{dom sigFnc} \land n \in \text{ran sigFnc}) \Rightarrow (n = (x \times x))
\]

\[
\forall x: \mathbb{Z} \bullet m \geq 10 \land m < 50 \land ((x, m) \in \text{dom sigFnc} \land n \in \text{ran sigFnc}) \Rightarrow (n = 30)
\]

\[
\forall x: \mathbb{Z} \bullet m \geq 50 \land ((x, m) \in \text{dom sigFnc} \land n \in \text{ran sigFnc}) \Rightarrow (n = (x \times x) \div (m \times x + 4))
\]

4.4.2.2.7. Formalization example of the triggers reactivity dimension:
Here \([EVENT]\) is the set of all events of a system. If the first event in a sequence of events is observed to be event \(e_1\), the next test sequence to be executed is \(\text{TestSeqNrM}\). Additionally the reactivity flag (ReactFlag), which lets the harness to record the fact that a reactive behaviour was captured, is set to true.

\[
\text{Test\_ReactToEvent}
\]

\[
\text{eventSeq: seq EVENT} \\
e1: EVENT \\
\text{TestSeqCurrent, TestSeqNrM: TESTSEQUENCE} \\
\text{ReactFlag: BOOLEAN}
\]

\[
(\text{head eventSeq} = e1 \Rightarrow (\text{TestSeqCurrent} = \text{TestSeqNrM} \land \text{ReactFlag} = T))
\]
4.4.2.3. **Formalization examples of the knowledge based scenarios reactivity dimensions:**

4.4.2.3.1. **Formalization example of the black spot based reactivity dimensions:**
The friction coefficient of the wheels with the road \((mju)\) is under a threshold \((mjuThreshold)\). The readings between the gyroscopic sensor in the car \((angleGyroscopic)\) and the steering wheel sensor \((angleSteeringWheel)\) differ by at least 15 degrees. The next test sequence to be executed is a sequence \((TestBrakeOnIce)\) which simulates the driver attempting an emergency brake on ice. Additionally the reactivity flag \((ReactFlag)\), which lets the harness to record the fact that a reactive behaviour was captured, is set to \textit{true}.

\[
\begin{align*}
\text{Test\_KnowledgeBasedScenario} \\
\text{TestSeqCurrent}, \text{TestBrakeOnIce}: \text{TESTSEQUENCE} \\
mju, mjuThreshold, angleGyroscopic, angleSteeringWheel: \mathbb{Z} \\
\text{ReactFlag}: \text{BOOLEAN} \\
(mju < mjuThreshold \land \left((angleGyroscopic > angleSteeringWheel \land (angleGyroscopic - angleSteeringWheel) \geq 15) \lor (angleGyroscopic \leq angleSteeringWheel \land (angleSteeringWheel - angleGyroscopic) \geq 15)\right)) \Rightarrow \\
(\text{TestSeqCurrent} = \text{TestBrakeOnIce} \land \text{ReactFlag} = T)
\end{align*}
\]

4.4.2.3.2. **Formalization example of the rare but safety important reactivity dimension:**
Here if the first test case in the current test sequence \((TestSeqCurrent)\) is the predesigned test case \(TestCaseNrN\) and the latest test result \((TestResult)\) is \textit{false} while the battery voltage \((BatteryVoltage)\) is under 2 volts, the next test sequence to be executed is \(TestSequenceNrL\). Additionally the reactivity flag \((ReactFlag)\), which lets the harness to record the fact that a reactive behaviour was captured, is set to \textit{true}.

\[
\begin{align*}
\text{Test\_KnowledgeBasedScenario} \\
\text{TestSeqCurrent, TestBrakeOnIce: TESTSEQUENCE} \\
mju, mjuThreshold, angleGyroscopic, angleSteeringWheel: \mathbb{Z} \\
\text{ReactFlag: BOOLEAN} \\
(mju < mjuThreshold \land \left((angleGyroscopic > angleSteeringWheel \land (angleGyroscopic - angleSteeringWheel) \geq 15) \lor (angleGyroscopic \leq angleSteeringWheel \land (angleSteeringWheel - angleGyroscopic) \geq 15)\right)) \Rightarrow \\
(\text{TestSeqCurrent} = \text{TestBrakeOnIce} \land \text{ReactFlag} = T)
\end{align*}
\]
Rationale Behind Step 2:

a. In addition to the rationale points a-d presented in Step 1, test reactivity when captured formally, can also be used to constrain the test execution paths such as in the cases of test generation performed via state machine navigation techniques.


The proposed usability model focuses on test generation based on equivalence partitions. While CTM_{CONTROL} is clearly the focus of this thesis, this part of the FRTP can be used with any test approaches based on the equivalence partitions principle.

In the proposed usability model, the tests generated for the SUT come from two different sets:

1. Tests generated systematically via an equivalence partition approach.
2. Tests generated to capture the desired test reactivity behaviour.

Which tests are selected to be run and from what combination of the above mentioned sets, depends on the test objectives and on what fault model/s is/are deemed more appropriate for the project at hand.
I. Tests generated systematically via an equivalence partition method.

Typically, testers base their test generation strategy on identifying a relevant and feasible (not necessarily explicit) fault-model (Burton 2002), which identifies the type of potential errors that can be detected by applying a particular type of tests. One of the challenges for testers is to provide the appropriate fault models against which the software should be tested. Making this fault model explicit helps to have a clear picture on the type of faults the testing process is assumed to detect as well as on the assumptions made on how these faults will be found.

Equivalence partition testing is a technique where the exhaustive test set is subdivided in subsets where the SUT is assumed to behave uniformly. In the usability model the partitioning heuristics are specified formally. This helps to make the uniformity hypothesis and any assumptions made explicit. Furthermore, this could be used for verification activities as well as to analyse possible links between the chosen partition heuristics and effectiveness of the testing process.

Furthermore, each equivalence partition is complemented with a weighting factor and a signal function to represent respectively the ‘density’ of the parameter values taken from that partition and the mapping of these values in continuous or discrete time. The tester chooses the best fitting time-patterns (such as continuous time, after time \( t \), periodic time, etc) and maps the test data values generated via the equivalence partitioning approach into the selected time patterns. Providing this way the functions for the test signals in each partition. The default signal function (used if no other is specified) is a simple linear one: \( y = m*t + b \).

For illustration purposes let us\(^{35}\) consider the velocity of a car, which has a possible value range of \([0, 200]\) km/h. Abstract partitions that can be considered are: Minimum (MinVel), Low (LowVel), Medium (MedVel), High (HighVel), and Maximum (MaxVel). The partitions can be

\(^{35}\) The plural “us” here denotes the author of the thesis and the reader.
parameterized into: \{[0-1), [1-30), [30-60), [60-200), [200-201)\} and the actual test data are generated by selecting at least one value from each partition. Or as in the example below, the weighting factor of 2 for the interval [0-1) indicates that the interval should be represented by two values. The signal function \( S_f = 0.9 \times x \) indicates that the data signal has the shape of a simple linear slope. It is assumed that the SUT will behave uniformly for all the remaining data in that partition.

In tabular format the partitions can be represented as in Table 25.

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>MinVel</th>
<th>LowVel</th>
<th>MedVel</th>
<th>HighVel</th>
<th>MaxVel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [km/h]</td>
<td>[0-1)</td>
<td>[1-30)</td>
<td>[30-60)</td>
<td>[60-200)</td>
<td>[200-201)</td>
</tr>
<tr>
<td>Weighting factor and signal function</td>
<td>( W=2; )</td>
<td>( W=3; )</td>
<td>( W=3; )</td>
<td>( W=3; )</td>
<td>( W=2; )</td>
</tr>
<tr>
<td>( S_f=0.9\times x )</td>
<td>( S_{f_1}=14\times t+1, )</td>
<td>( S_{f_2}=5\times t+10, )</td>
<td>( S_{f_3}=5\times t+10 )</td>
<td>( S_f= ) step function</td>
<td>( S_f= ) step function</td>
</tr>
<tr>
<td>Representative data selection [km/h]</td>
<td>{0, 0.9}</td>
<td>{1, 15, 29}</td>
<td>{30,45,59}</td>
<td>{60, 100,199}</td>
<td>{20, 201}</td>
</tr>
</tbody>
</table>

The same partitions can be formalized in Z as below:

Here the partitions are \( MinVel \) for velocities \( i \in [0-1) \) km/h; \( LowVel \) for velocities \( j \in [1-130) \) km/h; \( MedVel \) for velocities \( k \in [30-60) \); \( HighVel \) for velocities \( l \in [60-200) \); and \( MaxVel \) for velocities \( m \in [200-201) \).

\[
\begin{align*}
\text{Formal Partitions Example} \hline
\text{MinVel, LowVel, MedVel: } & \forall \text{ VELOCITY} \ni \text{MinVel} \rightarrow i \geq 0 \wedge i < 1 \\
\text{HighVel, MaxVel: } & \forall \text{ VELOCITY} \ni \text{HighVel} \rightarrow l \geq 60 \wedge l < 200 \\
\forall i: \text{ VELOCITY} \ni i \in \text{MinVel} \rightarrow i \geq 0 \wedge i < 1 \\
\forall j: \text{ VELOCITY} \ni j \in \text{LowVel} \rightarrow j \geq 1 \wedge j < 30 \\
\forall k: \text{ VELOCITY} \ni k \in \text{MedVel} \rightarrow k \geq 30 \wedge k < 60 \\
\forall l: \text{ VELOCITY} \ni l \in \text{HighVel} \rightarrow l \geq 60 \wedge l < 200 \\
\forall m: \text{ VELOCITY} \ni m \in \text{MaxVel} \rightarrow m \geq 200 \wedge m < 201
\end{align*}
\]
2. **Tests generated to capture the desired test reactivity behaviour.**

Test reactivity can be captured by designing a series of conceptual states to which the test process should react to. The term *conceptual state* here denotes one from all of the possible configurations of the SUT.

A conceptual state could be captured by solving a deterministic set of constraints over all the possible execution paths of the SUT. In this sense, reactive test behaviour decides the trajectory which the system takes among all of the possible test paths.

Hence, when test generation is performed on the fly, test reactivity is one of the test-selection criteria. For instance if after the execution of test *x*, the system’s behaviour fits a test reactivity dimension of interest, the testing process (i.e. which test is generated and executed next) responds to it. This analytically derived conclusion modifies the taxonomy of model based testing proposed by (Utting et al. 2006) and later (Zander-Nowicka 2009) as shown in the figure below (Fig. 22).
The FRTP is only concerned with having test reactivity captured formally. How the reactive tests are generated is an implementation issue and it is left to the tester to choose the approach that better suits her/him or the SUT. Section 5.2.2.2.2 of my thesis describes how I have implemented test reactivity in the M/S/S environment via constraint logic programming in Simulink.
Rationale Behind Step 3:

a. In addition to the reasons discussed in the previous steps, formally specifying the partitioning heuristics helps to make the uniformity hypothesis and any assumptions made explicit. Furthermore, this could be used for verification activities as well as to analyse possible links between the chosen partition heuristics and effectiveness of the testing process.

4.4.4. Step 4: Check Compliance to Test Objectives.

The testing process is stopped when the test objectives are met. Having the test objectives captured in formal notation reduces the risk of misunderstandings.

Adequate data coverage achieved through equivalence partition of data techniques does not necessarily also deliver adequate model coverage and it is here where the formalisation of the approach fully delivers its added value. This is due to the fact that test sequences which improve model coverage can be created automatically via pre-existing approaches developed for test generation from Statecharts.

Based on the specified model coverage criteria and the structural characteristics of the SUT, the tester can choose to generate tests via algorithms such as those of the Wp Method, All States, All Transitions, Unique Identifying Sequences, etc.
4.5. Reflections on the Chosen Research Methodologies

In this section, the research methodologies used for the proposed CTM\textsubscript{CONTROL} and the FRTP are discussed separately.

4.5.1. Research Methodology for CTM\textsubscript{CONTROL}

This research is based on a positivist philosophy and is aligned with the thinking that reality is objective and can be described by measurable properties, which are independent of the observer and her instruments (Myers 1997). A number of aspects of positivism as a research paradigm have been discussed by a number of researchers including (Myers 1997, Orlikowski and Baroudi 1991, Straub et al. 2005, Easterbrook et al. 2008). Of particular interest in scoping the philosophy of this research were the characteristics of positivist research elucidated by (Orlikowski and Baroudi 1991). These include characterizing a positivist research as: research with formal propositions; research with variables that can be quantifiably measured; and research that test hypotheses. The above, make the best case of why a positivist philosophy is the best fit for this research.

This research is also very much interested in the “analysis of quantitative analysis techniques, quantitative data using positivist mathematical techniques”, which is how Straub and colleagues (2005) describe quantitative research.

After scoping this research’s method as a quantitative one, and analysing it a finer granularity it became clear that this research is based on testing a hypothesis. More specifically, the rationale behind the proposed CTM\textsubscript{CONTROL} is based on the hypothesis that the classic CTM does not cover some of the control logic of the SUT, while CTM\textsubscript{CONTROL} does deliver better structural coverage.

Supporting or falsifying this hypothesis involves measuring the structural coverage delivered by CTM\textsubscript{CONTROL} as compared to CTM. For this, a
controlled experiment, which is used to test this hypothesis, was found to be a natural fit.

As is described in the next chapter of this thesis, a controlled experiment is constructed to test a SUT captured via a Statechart. The tests are initially generated via the classic CTM approach and the achieved structural coverage is measured. Then the tests are generated via the MC/DC aspect of CTM\textsubscript{CONTROL} and the achieved structural coverage is measured. During testing, the SUT is forced to remain in the same transition.

The independent variable of the controlled experiment is the methodology via which the tests are generated. The controlled variable in the experiment is the one transition where the SUT is forced to stay. The dependent variable is the structural coverage achieved by the two different testing methods. The dependent variable is measured at the end of all the tests generated via the CTM method and the second time after all the tests generated via the MC/DC aspect of the CTM\textsubscript{CONTROL} are executed.

In addition to the controlled experiment, it was decided that the research would benefit from observing CTM\textsubscript{CONTROL}’s behaviour in its ‘natural context’, while exercising an automotive SUT. In order to accommodate the quantitative nature of the approach and the need to capture some of the specifics of context, it was concluded that a logical methodological approach would be to translate the experimental logic into a case study approach.

The case study investigates experimentally the structural coverage delivered by CTM\textsubscript{CONTROL} as compared to CTM when used to test an Adaptive Cruise Control (ACC) used in cars. The ACC case study is used to support or falsify the CTM\textsubscript{CONTROL} hypothesis based on deductive reasoning. The reasoning is based on the measured results of the experimental approach at the core of the ACC case study.

The ACC is chosen as a system, which has attracted the interest of a number of researchers including (Zander-Nowicka 2009, Philipps et al. 2003, Jiang
et al. 2007) and is used in several case studies and experiments in the domain of automotive software testing.

The case study was selected analytically for two reasons:

1. It is an automotive case study, hence allows observing CTM\textsubscript{CONTROL}’s behaviour in its natural context.

2. The ACC is one of the canonical examples of the subject-matter. Hence it has the potential to both facilitate understanding and to be generalized via abductive reasoning by other researchers. The last argument is based on what Stake calls the concept of naturalistic generalisation (Stake 1995). This in the hope that future work from other researchers will achieve improvements on the results reported in this thesis.

4.5.2. Research Methodology for the Formalization of a Reactive Testing Process and Taxonomy Contributions

The validation of the FRTP and its taxonomies is outside the scope of this thesis.

The FRTP was designed via a conceptual abstraction approach informed by literature. The design followed an iterative approach. The main sources that informed the design of the FRTP are the QA standards regulating the field of automotive software and the literature on automotive software testing.

A partial usability testing of the FRTP is done\textsuperscript{36} via illustrating the use of CTM\textsubscript{CONTROL} within it.

The test reactivity taxonomy was built in a bottom-up approach. The work started with identifying in literature the different types of test reactivity, which are important to test embedded automotive software. While there is a number of researchers that have focused on reactive testing, one of the more comprehensive descriptions of the type of test reactivity can be found in (Zander-Nowicka 2009). After elucidating the types of reactive testing, they

\textsuperscript{36} In APPENDIX B of this thesis.
were classified following an iterative approach. In each iteration, using a combination of walk-throughs and usability testing methods, the classifications were refined guided by three criteria:

4. Does all the elucidated content fit (i.e. could all the types of test reactivity be classified via the taxonomy)?
5. Can any of the classifications be merged with each other?
6. Can any of the hierarchical relationships be flattened?

The proposed changes in the model-based taxonomy are informed by the literature review in automotive testing and the realities of reactive testing.

A validation via a usability testing approach is envisaged to be the method that best suits their intended use.

4.6. Conclusions

This chapter proposes a new testing method tailored to the requirements of safety-related embedded automotive software. Informed by the industry’s reluctance of adopting ‘yet-another-method’, it focuses into proposing a practical solution, which makes use of methods and toolsets that are already familiar for the automotive software tester. Aiming to provide a gentle learning curve for the testers, the proposed CTM\textsubscript{CONTROL} extends the Classification Tree Method that is a classic and popular testing method, which is routinely used in industry.

CTM\textsubscript{CONTROL} has a particular focus on the specification-based control-logic of the system-under-test. CTM\textsubscript{CONTROL} brings the CTM closer to the quality assurance requirements of the latest standards, which regulate automotive software. It does so by targeting the Modified Condition/ Decision Coverage objective set out from these standards.

In response of the QA requirements on formality (ISO 2011), this chapter proposes the Formalization of a Reactive Testing Process. Additionally it proposes enhancements on a previously published model-based testing
taxonomy (Utting et al. 2010, Zander-Nowicka 2009) and a new taxonomy for reactive testing.

The FRTP links reactive testing to the field of formal testing and aims to facilitate the incorporation of formality in the reactive testing process.

The TRT helps the tester to elucidate the types of test reactivity that might be beneficial for the test process at hand. The formalization examples of the TRT dimensions facilitate the use of the FRTP and in more general terms they help the formalization of test reactivity.

CTM\textsubscript{CONTROL}, the Formalization of a Reactive Testing Process and the taxonomic contributions are all tailored for the field of safety-related automotive software. They have been developed with a view to facilitate their use in industry.

From a methodological point of view, the chapter arrives at the conclusion that the hypothesis at the core of the proposed CTM\textsubscript{CONTROL} can be tested via a controlled-experiment approach and further investigated via an automotive case study.
PART 3
Validation, Evaluation and Conclusions
5. Experimental Evaluation

“What it [science] is, is a problem which I set for myself …. After some time, I was reminded of a little poem:

A centipede was happy quite, until a toad in fun
Said, “Pray, which leg comes after which?”
This raised his doubts to such a pitch
He fell distracted in the ditch
Not knowing how to run.

All my life, I have been doing science and known what it was, but what I have come to tell you—which foot comes after which—I am unable to do, and furthermore, I am worried by the analogy in the poem that when I go home I will no longer be able to do any research.”

Richard Feynman, 196937.

This chapter presents the validation of the proposed CTM\textsubscript{CONTROL} via a controlled experiment approach. The experimental evaluation continues with an automotive case study, which compares CTM and CTM\textsubscript{CONTROL} while testing an Adaptive Cruise Control (ACC) system. Included in the case study section, is the prototypical implementation of CTM\textsubscript{CONTROL} in the M/S/S environment. CTM\textsubscript{CONTROL} also illustrates the proposed FRTP.

5.1. The Controlled Experiment

As discussed in Chapter 4 of this thesis, CTM while providing a clear strategy for systematic testing, gives the engineer no control over the degree of model coverage and can leave parts of the control logic of the model unexercised. Embedded systems often have logical statements guarding their transitions and guards and these will not be necessarily exercised by a classical requirement-based test generation approach. CTM, in this, behaves in line with the typical requirement based testing approach and does not

37 The author of the poem remains anonymous. Feynman quoted it when answering the question “What is science?” while addressing the National Science Teachers Association in New York City. The speech appeared in print in The Physics Teacher, (7), 6, 1969, pp. 313-320.
provide assurances regarding the presence or lack of unintended functionality in the SUT.

To validate the proposed concept, I constructed a controlled experiment, which investigates some of the differences in structural coverage achieved by the proposed CTM\textsubscript{CONTROL} compared to the classic CTM.

The model built for the controlled experiment in the M/S/S environment is shown in Figure 23. As part of my strategy to ensure that there are no unintended contributors to the experiment’s result, the experiment isolates one single transition \( R \) for analysis (Fig. 24).

![Figure 23: Overview of the Controlled Experiment Model](image)

As part of this strategy, the SUT (Fig. 24) has four states that do not manipulate data, and, during simulation the guards of all the transitions bar \( R \) are kept constant to a value that causes the system to remain in the transition of interest (\( R \) is evaluated in every simulated second).
The test signal values for the guards of the other transitions are depicted in Figure 25.
Figure 25: Tests signals, designed to isolate the transition of interest.

$R$ is guarded by a logical expression (2), which has been abstracted from statements, which are often found in embedded software systems:

$$A==3 \text{ AND } B==1 \text{ OR } C-A< 1$$

Here $A= [0 \ 1 \ 2 \ 3 \ 4]$, $B= [0 \ 1 \ 2]$ and $C= [0 \ 1 \ 2 \ 3 \ 4]$ are the input parameters of interest and the respective possible values they can take.

The fact that the input parameters can only have a limited set of discrete values (rather than a range of analogue values) is part of my strategy to construct an experiment with a slight positive bias towards the classic CTM. This, since the limited set of discrete values allows including each possible value in the test signal. Whereas, if the parameters could take any values
from a possible range, the test signal would be generated by selecting just one value from each partition of the range. Consequently, it could be argued that the actual choice of classification classes or values to represent them, could potentially make a difference to the strength of CTM’s structural coverage. Being in the position to be able to include all the possible values of the input parameters, allows investigating the maximal cover that the classic CTM approach can deliver for this model.

The classification classes and the values representing them for the classic CTM approach are depicted in Figure 26.

To generate the tests for the MC/DC aspect of CTM$_{\text{CONTROL}}$, all the atomic logic expressions in (2), are considered separately and each decision and condition is toggled between true and false. The atomic logic expression $A = 3$, can be toggled between true and false respectively for the values of $A = [3 \ 0]$. Similarly, the atomic expression $B = 1$ can be toggled between true to false by the values of $B = [1 \ 0]$. For the selected values of $A = [3 \ 0]$, the expression $C - A < 1$ can be toggled between true and false by a number of combination of values, including the values of $C = [0 \ 1 \ 3 \ 4]$. These values
for A, B and C form the classification classes in the MC/DC aspect of the classification tree (Fig. 27).

The actual tests for the classic CTM and the MC/DC portion of the CTM_{CONTROL} are generated automatically. The logic of the combinations algorithm used to generate tests by combining the different parameter values for both approaches is delineated in Figure 28.
The algorithm returns all the combinations of the vector elements passed into it, organized into a matrix structure. For example, all the combinations of the elements of \( X = [1 \ 2 \ 3], \ Y = [4 \ 5], \ Z = [6 \ 7 \ 8] \), are structured in the following matrix 
\[
\begin{array}{ccc}
1 & 4 & 6 \\
1 & 4 & 7 \\
1 & 4 & 8 \\
1 & 5 & 6 \\
1 & 5 & 7 \\
1 & 5 & 8 \\
2 & 4 & 6 \\
2 & 4 & 7 \\
2 & 4 & 8 \\
2 & 5 & 6 \\
2 & 5 & 7 \\
2 & 5 & 8 \\
3 & 4 & 6 \\
3 & 4 & 7 \\
3 & 4 & 8 \\
3 & 5 & 6 \\
3 & 5 & 7 \\
3 & 5 & 8 \\
\end{array}
\]

Figure 28: The combinations algorithm

The test signals generated for the experiment via the CTM approach are depicted in Figure 29. The test signals generated via the CTM\_CONTROL approach are depicted in Figure 30. In both figures the yellow line depicts the test signal generated for parameter \( A \), the magenta line represents the test signal for \( B \) and the cyan colored line represents \( C \).
Figure 30: Test signals generated via the MC/DC aspect of the CTM$_{\text{CONTROL}}$ approach
5.1.1. The Coverage Reports

The test harness built for the controlled experiment, can be executed in CTM mode or in MC/DC mode (the latter generates the tests for the MC/DC aspect of CTM\textsubscript{CONTROL}).

The SUT has been implemented via a Stateflow Statechart (MathWorks 2008). As shown in the coverage report (Fig. 31), when running the test harness in the CTM mode, the only logical condition in \(R\) for which the test signal reversed the outcome is that of \(A=3\) and hence only a 33\% MC/DC coverage is achieved.

![Figure 31: Coverage achieved when using the classic CTM method.](image)

When the system is executed in the MC/DC mode of the CTM\textsubscript{CONTROL}, as shown in the coverage report (Fig. 32), all the possible combinations of the logical coverage are exercised from the test signal and a 100\% MC/DC coverage of the transition \(R\) is achieved.
Figure 32: Complete coverage of the logical statements inside the transition

While achieving 100% MC/DC coverage in an MC/DC approach is not unexpected, a 33% rate of coverage with the classic CTM shows that a whole category of errors in that transition would not be unearthed by its tests.

The results of the controlled experiment are analyzed in the next chapter (Chapter 6) of my thesis alongside the results from the automotive case study, which is discussed next.
5.2. An Automotive Case Study: The Adaptive Cruise Control

This case study focuses into the structural coverage analysis for tests generated via CTM\textsubscript{CONTROL} and CTM, when used in the testing process of an automotive system. The SUT used in this case study is an automotive ACC. The ACC is chosen as a system, which has attracted the interest of a number of researchers (including (Zander-Nowicka 2009, Philipps et al. 2003, Jiang et al. 2007)) with a view to facilitate understanding and comparing results with other researchers from the area.

The case study also illustrates how CTM\textsubscript{CONTROL} can be used within the FRTP proposed in Chapter 4.

A roadmap that shows a ‘bird’s-eye’ view of the structure of the information presented in this section and of some of the possible ‘routes’ to read this case study is depicted in Figure 33.
A partial specification for the ACC written in natural language is given in Section 5.2.1 and the reader is directed to ANNEX A to refer to a partial specification in Z for the ACC. The implementation of the prototype is discussed in Section 5.2.2.

The use of CTM\textsubscript{CONTROL} within the FRTP proposed in Chapter 4 is illustrated in an abbreviated version in Section 5.2.3 and the full version in APPENDIX B. A structural coverage analysis for tests generated via CTM\textsubscript{CONTROL} as compared to tests generated via the classic CTM is presented in Section 5.2.4.
5.2.1. Partial Specification for the ACC

The ACC keeps the velocity of a vehicle to a value chosen by the driver while maintaining a safe distance from any physical obstacles in its path (Fig. 34).

![Schematic overview of an ACC (FORD 2012)](image)

In this case study, the ACC’s interface with the driver is a lever, which can be in one of five positions: Off, Neutral, Set+, Set- and Resume (Fig. 35).

![ACC interface](image)

On command from the driver, the ACC takes control of the velocity of the vehicle. The driver can set the required velocity via the Set+/Set- positions, which increment/decrement the velocity by a predetermined value (in this
example by 5 km/h (1.4 m/s)). The car also has radar technology and if an obstacle is detected in front of the car, it adjusts the speed so as to be at a safe distance from the obstacle in front. When the deceleration needed to avoid a detected obstacle goes beyond a safe value an auditory alarm warns the driver.

The ACC can be operated for any velocities up to 69.4 m/s and it can bring the vehicle to a full stop.

The driver must be able to increase the speed at any time by pressing the accelerator pedal, or reduce the speed by pressing the brake pedal. When the accelerator pedal is released, the ACC will regain control. Any time the brake pedal is pressed, the system must go inactive. Following this, when the brake is released and Set+, Set. or Resume is pressed, the ACC becomes active and treats the vehicles’ current velocity as the selected speed. However, if an ‘Off’ has occurred in the intervening time, the Set+, Set. and Resume commands do nothing.

### 5.2.2. Highlights from the Prototype Implementation

The prototype built for the ACC case study is understood as the ensemble of models that includes the Test Harness and the environment it communicates with (Fig. 36).

![Figure 36: Conceptual view of the ACC prototype](image)
5.2.2.1. The SUT
The ACC, which serves as the SUT is modelled via a Stateflow Statechart (Fig. 37) and its input and output parameters\(^{38}\) are listed in Table 26.

<table>
<thead>
<tr>
<th>Table 26: I/O Parameters of the SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input parameters</strong></td>
</tr>
<tr>
<td>InitSetSpeed [m/s]</td>
</tr>
<tr>
<td>TargetPresent [QDO](^{39})</td>
</tr>
<tr>
<td>LeverPos [QDO]</td>
</tr>
<tr>
<td>Step [m/s]</td>
</tr>
<tr>
<td>MaxAllowedVel [m/s]</td>
</tr>
<tr>
<td>MinAllowedVel [m/s]</td>
</tr>
<tr>
<td>PedalFlag [QDO]</td>
</tr>
<tr>
<td>Headway [m]</td>
</tr>
<tr>
<td>TargetVehicleVel [m/s]</td>
</tr>
<tr>
<td><strong>Output parameters</strong></td>
</tr>
<tr>
<td>Out_ActualVel [m/s]</td>
</tr>
<tr>
<td>Out_LeverPos [QDO]</td>
</tr>
<tr>
<td>DangerousBrakingAlarm [QDO]</td>
</tr>
</tbody>
</table>

\(^{38}\) Local parameters of the Statechart have been omitted from the description for ease of reading.

\(^{39}\) QDO in this document stands for Quantity of Dimension of One, which is a quantity which has no associated physical dimension and no measuring unit.
ahead, produces a deceleration, which is unsafe for the vehicle (1 = alarm on; 0 = alarm off).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out_ActualPedalFlag [QDO]</td>
<td>Value of the pedal flag at the exit of the Statechart (0 = no pedal pressed; 1 = acceleration pedal pressed; 2 = brake pedal pressed).</td>
</tr>
<tr>
<td>NotAllowedVelFlag [QDO]</td>
<td>This parameter is set to 1 if the velocity of the vehicle is outside the allowed values where the ACC can be operational, otherwise is set to 0.</td>
</tr>
<tr>
<td>Out_TargetVelocity [m/s]</td>
<td>The velocity of the vehicle ahead (if any) measured when exiting the Statechart.</td>
</tr>
</tbody>
</table>

The SUT has four states: ‘Off’, ‘Standby’, ‘CruiseByVelocity’ and ‘CruiseByDistance’. The state ‘Off’ can only be reached from the state ‘Standby’, while the other three states can all communicate with each other.

Figure 37: The SUT Statechart
CHAPTER 5 EXPERIMENTAL EVALUATION

All the states manipulate data and this functionality is programmed in Stateflow, which uses the following notation ⁴⁰ (Fig. 38):

All the transitions, bar the default one (taken when entering the Statechart), are guarded via labels written according to the notation depicted in Figure 39.

---

| Name of the State/entry: actions executed when entering the state | exit: actions executed when exiting the state |
---|---|

**Figure 38: Notation used inside the SUT’s states**

---

`event [condition][condition_action]/transition_action`

Where:

- `event`: specifies an event that causes the transition to be taken, provided the condition, if specified, is true.
- `condition`: specifies a boolean expression that, when true, validates a transition to be taken for the specified event. It is enclosed in square brackets `[]`.
- `condition action`: A condition action follows the condition for a transition. It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed. It is enclosed in curly braces `{}`.
- `transition action`: The transition action is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. It is preceded with a backslash.

**Figure 39: Notation used in the SUT’s transitions (MathWorks 2003, Pretschner et al. 2005, Conrad and Fey 2005)**

---

The SUT’s transitions manipulate data within the transition action element of their labels and for ease of reading, the labels alongside their respective source and target states are numbered and depicted in Table 27.

Table 27: SUT Transition Labels

<table>
<thead>
<tr>
<th>No</th>
<th>Off</th>
<th>Brandly</th>
<th>CruiseByVelocity</th>
<th>CruiseByDistance</th>
<th>Transition Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>LevelPos=1</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>LevelPos=1</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>LevelPos=1</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>LevelPos=0</td>
<td></td>
<td>LevelPos=1</td>
</tr>
</tbody>
</table>

The manipulation of data within the states and during the transitions is implemented via Graphical Functions, which are functions defined graphically and use Stateflow’s action language (MathWorks 2003).

The SUT implementation includes two graphical functions: `funcCalcVel()`, which is used by the ACC to modify the vehicle’s velocity when there is no obstacle detected ahead, and `funcCalcDistance()`, which is used to modify the velocity when there is a (potential) obstacle detected ahead.

5.2.2.1.1. Graphical Function `funcCalcVel()`

When the ACC’s radar does not detect any obstacles (such as other vehicles) in the path, it keeps the vehicle’s velocity at the value required by the user. This functionality has been implemented as depicted in Figure 40 and is used by the state ‘CruiseByVelocity’ as well as the transitions 4, 6 and 10 (Table 27).
5.2.2.1.2. Graphical function funCalcDistance()

When the ACC’s radar detects a vehicle ahead that is travelling at a slower speed it calculates the needed deceleration according to the braking distance that the car can achieve in the road’s conditions.

The braking distance can be calculated as a function of the vehicle’s initial speed $V_0$ and the coefficient of friction between the tires and the road $\mu$:

$$D = \frac{V_0^2}{2g\mu}$$

Where:

- $D$ = Braking distance
- $g$ = Acceleration due to gravity (9.8 m/sec$^2$)
- $V_0$ = Vehicle’s initial speed
- $\mu$ = Coefficient of friction between the tires and the road.
The values for kinetic friction can vary from somewhere in the region of \( \mu = 0.2 \) for tyre on snow, to around 0.8 for tyre in excellent condition on dry road. If the deceleration needed goes beyond a value, which can be safely delivered by the vehicle, the ACC warns the driver via an auditory alarm.

The logic of how this feature has been implemented is described below in pseudo code (Fig. 41):

\[
\Delta V = V_0 - V_{obstacle}
\]

\[
if \quad \Delta V > 0
\]

\[
D = \frac{V_0^2}{2g\mu}
\]

\[
t = \frac{D}{V_0}
\]

\[
a = \frac{\Delta V}{t}
\]

\[
if \quad a \leq \text{safe value}
\]

\[
V' = V_0 + at
\]

\[
else
\]

\[
\text{warn driver}
\]

Figure 41: Rationale behind brake by distance

The actual implementation in the form of a graphical function is depicted in Figure 42.
5.2.2.2. The Test Harness
The test harness for testing the ACC was implemented in the M/S/S environment. During the research process which led to this PhD thesis, different versions for the test harness were designed and developed. Among the different implementations, a number of different approaches of implementing a classification tree were devised. The approaches were implemented via graphical Simulink blocks or code written in MATLAB’s own programming language. The next section describes a graphical implementation for classification trees.

5.2.2.2.1. Graphical Implementation of a classification tree
One of the core elements of the graphical implementation of classification trees is depicted in Figure 43. The PartitionPatern is a (reusable) pattern block programmed in Simulink, which allows to mask out any signal values that fall outside a user-specified range.
In other words, for a signal which represents the actual values of an input parameter for a SUT, the \textit{PartitionPattern} block filters out all signal values outside the user defined \textit{min\_val} and \textit{max\_val}. The scope depicted in \textit{Figure 44} shows how the input signal has been filtered to only output those values, which fall between \textit{min\_val}= 30 and \textit{max\_val}=60.

The actual implementation of the data partition pattern is detailed in \textit{Figure 45}.
This pattern block can be used to build a full classification tree. For instance, Figure 46 illustrates the implementation of a classification tree, which has three classifications.
The scope in Figure 47 shows how the model depicted in Figure 46 automatically generates three user defined partitions from an input signal.
While the implementation approach (whether it is graphical or not) does not influence the coverage capabilities of CTM\textsubscript{CONTROL}, for this case study for ease and brevity of explanation, all the classification trees and test generation discussed from this point on are implemented via algorithms written in MATLAB’s own programming language\textsuperscript{41}.

5.2.2.2.2. Highlights from the Test Harness’ implementation

In this document\textsuperscript{42}, the term Test Harness (Fig. 48) refers to the ensemble of models that include: the Test Generator, which controls the test process (i.e. decides which test sequences are ran); the SUT; and, the Test Verdicts model, which operates as an oracle.

The Test Generator’s behaviour reacts to the Testing Mode selected by the user (i.e. with or without MC/DC coverage); to whether reactive testing has

\textsuperscript{41} It was felt that including only the non-graphical implementation for classifications from this point on, helps to produce a more compact document and to keep the focus on the main contributions reported in this thesis.

\textsuperscript{42} Where the description of important elements refers to models and/or sub models, which are visually too complex to allow for a natural flow of reading, the ideas are explained via abstract/conceptual diagrams. A selection from the original diagrams of the system implementation can be found in the ANNEX B of my thesis.
been selected; to a number of outputs from the SUT (including reactivity flags) and to outputs from the Test Verdicts.\textsuperscript{43}

The functionality of the \textit{Test Generator} is delivered by a number of nested and layered models depicted in \textit{Figure 49}.

\textit{Figure 48: A view of the test harness for the ACC}

The Test Verdicts' model is fully operational when there are \textit{reference signals} available. Here the term \textit{reference signal} refers to a signal, which is used for comparison with the output signal from the SUT (the expected values for the SUT outputs). While the prototype implementation, allows the user to use a reference signal, its use is outside the scope of my research hence it is not discussed further in my thesis. It is mentioned in this footnote based on the rationale that it could be of interest if this \textit{Test Harness} is used in industry.
In the top level inside the Test Generator, reside the Test Sequence Selector model and the Test Signal Creation Layer model.

If some desired reactive behaviour is detected, the Reactive Testing Control Layer selects the test sequence which should be ran next.

The reactive behaviour scenarios of interest are implemented as models which are executed only when a relevant reactive behaviour is observed (depicted as “Models that capture the reactive behaviour” in Figure 49). Figure 50 depicts a simple example of ‘capturing’ an observed behaviour which automatically triggers reactivity. In this example, if the vehicle’s velocity is above 40 [m/s] and the radar detects a vehicle ahead which is traveling at a velocity lower than 17 [m/s], the Test Sequence Selector Reacts and chooses a test sequence which has been designed to follow this observed scenario.
Figure 50: The implementation of the reactive test abstract state

Overall, it is the Test Mode Control Layer (Fig. 51) that makes the final decision on which tests to execute next. If reactive testing behaviour was detected, it ‘obeys’ the Reactive Testing Control Layer’s decision. Otherwise, it decides on the next testing sequence based on which testing mode has been selected (i.e. with or without MC/DC coverage). This decision which happens inside the Sequence Selector, is passed then to the Test Signal Creation Layer.

Figure 51: Inside the Test Mode Control Layer

The Test Signal Creation Layer executes the decision. To execute the decision, the Test Signal Creation Layer (Fig. 52) identifies the required test data signals; it then loads up the appropriate test data signals and after assigning them to the correct output ports, exercises the SUT.
The Test Signal Creation Layer has been designed to be able to load up test data signals that are generated in three different approaches:

1. Generated automatically on-the-fly via batch files written in MATLAB’s programming language.
2. Generated automatically on-the-fly based on input from a Graphical User interface (GUI).
3. Data generated offline (manually or automatically) and logged in an environment or toolset which can connect to the M/S/S environment. Microsoft Office Excel for instance is one of the tools that has been connected to this Test Harness.
5.2.3. **Illustration of Using CTM\_CONTROL within the Formalization of a Reactive Testing Process**

The Formalization of a Reactive Testing Process proposed in Chapter 4 of this thesis, assumes the existence of a formal specification for the SUT, hence a (partial) formal specification for the SUT, written in Z notation, is presented in APPENDIX A of my thesis.

The workflow of the FRTP starts with capturing formally the test objectives. It follows with the formalization of the test reactivity, partition heuristics, and the formalization of test data generation and selection\(^{44}\).

5.2.3.1. **Formalization of Test Objectives**

The test objective for this case study is to measure the MC/DC coverage delivered by CTM\_CONTROL in comparison to that delivered by the classical CTM. This objective can be captured formally in Z notation. As specified in the Z schema below, the test harness can be executed in two modes: CTM or CTM\_CONTROL. Here coverage\(_A\), coverage\(_B\) and rep\(!\) are coverage reports which provide the percentage of MC/DC coverage achieved by the testing process. The Boolean rep\(!\) is set to true if the CTM\_CONTROL mode achieves a higher MC/DC coverage than the CTM mode. Otherwise rep\(!\) is set to false.

\[
\text{TESTMODE} ::= \text{CTM} \mid \text{CTM\_CONTROL}
\]

\[
\begin{align*}
\text{\_ACCTestHarness} \quad & \\
\text{testMode} : \text{TESTMODE} \quad & \\
\text{coverage\(!\)} : \mathbb{Z} \quad & \\
\text{coverage\(_A\), coverage\(_B\)} : \mathbb{Z} \quad & \\
\text{rep\(!\)} : \text{BOOLEAN} \quad & \\
\text{testMode} = \text{CTM} \Rightarrow \text{coverage\(!\)} \neq \text{coverage\(_A\)} \quad & \\
\text{testMode} = \text{CTM\_CONTROL} \Rightarrow \text{coverage\(!\)} \neq \text{coverage\(_B\)} \quad & \\
\text{coverage\(_B\)} > \text{coverage\(_A\)} \Rightarrow \text{rep\(!\)} = \text{T} \quad & \\
\text{coverage\(_B\)} \leq \text{coverage\(_A\)} \Rightarrow \text{rep\(!\)} = \text{F} \quad &
\end{align*}
\]

\(^{44}\) Selection in this instance is understood as the choice of which test sequences are ran (i.e. test control).
5.2.3.2. **Formalization of Test Reactivity**

In the proposed test reactivity taxonomy, as depicted in *Figure 53*, test reactivity dimensions fall into three major categories:

1. Coverage criteria related.
2. Input, output and temporal space related.

The test reactivity, which was considered in this case study falls into the *Signal Properties* of the *Input, output and temporal space related* dimension of the proposed test reactivity taxonomy (*Fig. 53*).

![Figure 53: Case study test reactivity mapped into the test reactivity taxonomy](image)

For the case study, two reactive scenarios were captured and implemented in the M/S/S environment.
Reactive Scenario No. 1: The vehicle is travelling at a speed \( (ActualVel?) \) of over 40 \([\text{m/s}]\) \((144 \,[\text{km/h}])\). The ACC is turned on and the radar picks up another vehicle that abruptly cuts in front of the first vehicle \((TargetPresent?=T)\). The other vehicle is travelling at a speed \((TargetVel?)\) lower than 17 \([\text{m/s}]\) \((61.2 \,[\text{km/h}])\).

The next test after this scenario is observed, should investigate what happens if the vehicle in front proceeds to brake abruptly and come to a full stop. In other words, the test harness should execute the test case designed to be triggered if this scenario (reactive conceptual state) is detected. The test case designed for this test scenario specifies that the first vehicle’s velocity should be 40 m/s \((NextActualVel!=40)\); the velocity of the vehicle in front should be 0 m/s \((NextTargetVel!=0)\); the ACC lever position should be in position 2 \((LeverPos=2)\); and, the driver should press the brake’s pedal \((PedalFlag=2)\).

---

\[
\begin{align*}
\text{ReactTestAbstState} &
\text{ActualVel?}, \, \text{TargetVel?}, \, \text{NextActualVel!}, \, \text{NextTargetVel!}: \text{VELOCITY} \\
\text{TargetPresent?}: \text{BOOLEAN} \\
\text{LeverPos}: \text{LEVERPOSITION} \\
\text{PedalFlag}: \text{PEDALFLAG} \\
\text{ActualVel}? &> 40 \land \text{TargetPresent}? = T \land \text{TargetVel}? < 17 \Rightarrow (\text{NextActualVel!}==40 \land \text{NextTargetVel!}==0 \land \text{LeverPos}==2 \land \text{PedalFlag}==2)
\end{align*}
\]

---

Reactive Scenario No. 2: The vehicle is travelling at a speed \( V_A \) \((ActualVel?)\) of over 40 \([\text{m/s}]\). The ACC is turned on and the radar picks up another vehicle \((TargetPresent?=T)\) that is travelling at a speed \( V_B \), which is around the same speed as that of our vehicle \((ActualVel?-1)\land TargetVel?<((ActualVel?+1))\)). The next test after this scenario is observed, should investigate what happens if our vehicle has matched the previous velocity of the vehicle in front \((NextActualVel!=TargetVel?)\), while the vehicle in front has accelerated and has increased its velocity by 10 m/s \((TargetVel?+10)\). Additionally the ACC lever should be in position 2 \((LeverPos=2)\) and the driver should not press any pedals \((PedalFlag=0)\).
5.2.3.3. Formalization of the Partition Heuristics, and Test Data Generation

The testing activities discussed here depict a point during the test process where the SUT parameters, which are considered as of interest are: *InitSetSpeed*, *TargetVehicleVel*, *TargetPresent*, *LeverPos* and *PedalFlag*.

The data partition heuristics are formalized and test data values are selected for each of these five parameters. The formalization is presented both in Z notation and via a tabular format for illustration purposes. The formalization is detailed for each parameter separately.

For the sake of brevity, here is detailed only the formalization of the data partitions and their parameterization for one of the parameters (*InitSetSpeed*). For completeness, the full illustration of the CTM\textsubscript{CONTROL} using FRTP is provided in *APPENDIX B* of this thesis.

### 5.2.3.3.1. Formalization of the data partitions and the parameterization of the partitions for *InitSetSpeed*.

The velocity of the current vehicle where the ACC under test is installed is denoted as *InitSetSpeed*. The data partitions for this parameter are *Min_Vel* for velocities $i\in[0-8)$ km/h; *Low_Vel* for velocities $j\in[8-14)$ km/h; *Med_Vel* for velocities $k\in[14-18)$; *High_Vel* for velocities $l\in[18-43)$; and *Max_Vel* for velocities $m\in[43-70)$. 
CHAPTER 5 EXPERIMENTAL EVALUATION

The partitions can also be presented in a tabular format as below (Table 28).

Table 28: InitSetSpeed (Vehicle Velocity)

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>Min_Vel</th>
<th>Low_Vel</th>
<th>Med_Vel</th>
<th>High_Vel</th>
<th>Max_Vel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [m/s]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weighting factor and signal function</td>
<td>W=1; Sf= interpolated</td>
<td>W=3; Sf= interpolated</td>
<td>W=1; Sf= interpolated</td>
<td>W=3; Sf= interpolated</td>
<td>W=1; Sf= interpolated</td>
</tr>
<tr>
<td>Data selection [m/s]</td>
<td>(0)</td>
<td>(8.6, 8.9, 9.2)</td>
<td>(16.7)</td>
<td>(41.9, 42, 42.2)</td>
<td>(69.4)</td>
</tr>
</tbody>
</table>

The MC/DC coverage for this case study is focused on the parameters *LeverPos*, *PedalFlag* and *TargetPresent* and their use in the state-transitions of the SUT. To generate the tests for the MC/DC aspect of CTM<sub>CONTROL</sub>, all the atomic logic expressions part of the state-transition guards, need to be considered separately. Each decision and condition is toggled between true and false. For example, the atomic logic expression LeverPos!=0, can be toggled between true and false respectively for the values of LeverPos!= (0, ¬0). Similarly, the atomic expression PedalFlag==2 can be toggled between true to false by the values of PedalFlag= (2, ¬2). The non-fully parameterized set of parameter values, which toggle all the atomic logic expressions of the transitions between true and false is depicted in Table 29:
Table 29: Non-fully parameterized set of values

<table>
<thead>
<tr>
<th>LeverPos</th>
<th>PedalFlag</th>
<th>TargetPresent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>not 0</td>
<td>not 2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>not 1</td>
<td></td>
</tr>
<tr>
<td>not 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One from the possible refinements of the above representation of data into a fully parameterized set of values is given in Table 30.

Table 30: The parameter values considered for the MC/DC

<table>
<thead>
<tr>
<th>LeverPos</th>
<th>PedalFlag</th>
<th>TargetPresent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All the test data generated from the process of parameterizing the CTM partitions and targeted MC/DC coverage for the case study are depicted in Table 31. In the cases when multiple data values have been chosen from the same data partition, these data are featured within brackets. For instance, the data values for Vehicle Velocity are (0, (8.6, 8.9, 9.2), (16.7), (41.9, 42, 42.2), (69.4)). The bracket (8.6, 8.9, 9.2) denotes the fact that the three values it contains, come from the same data partition.
5.2.3.4. Tests Generation for the CTM and the MC/DC Aspect

The tests for both the classic CTM and CTM\textsubscript{CONTROL} are generated automatically based on the test data values provided by the user (in this case the data presented in Table 31). The user can provide these values via a graphical user interface (GUI) (Fig. 55), or through passing them to scripts written in MATLAB. The test data can also be read directly from an Excel file (Fig. 54) or generated from other toolsets, which are connectable with the M/S/S environment.

```matlab
TestDataMatrix = xlsread('NameOfExcelFile','NameofExcelSheet','range')
```

Figure 54: Example of MATLAB syntax to read data from an Excel file

Figure 55 depicts part of the GUI, which allows the user to select the test mode, the inclusion of reactive testing and to type in the values to be used for test generation.

<table>
<thead>
<tr>
<th>Test data generated via a CTM approach</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle velocity</td>
<td>(0, 8.6, 8.9, 9.2), (16.7), (41.9, 42, 42.2), (69.4))</td>
</tr>
<tr>
<td>Vehicle in front velocity</td>
<td>(0, 8.6, 8.9, 9.2), (16.7), (41.9, 42, 42.2), (69.4))</td>
</tr>
<tr>
<td>Vehicle ahead detected flag</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>ACC lever position</td>
<td>(0, 1, 2, 3, 4)</td>
</tr>
<tr>
<td>Pedal pressed flag</td>
<td>(0, 1, 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test data generated via the MC/DC aspect of CTM\textsubscript{CONTROL}</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeverPos (ACC lever position)</td>
<td>(0, 1, 2, 3, 4)</td>
</tr>
<tr>
<td>PedalFlag (Pedal pressed flag)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>TargetPresent (Vehicle ahead detected flag)</td>
<td>(0, 1)</td>
</tr>
</tbody>
</table>
The test sequences for the CTM aspect are generated by using an all combinations algorithm (described in Figure 28) over the data values selected to parameterise the equivalence partitions (Table 31). The algorithm generates a M-by-N matrix $TestDataMatrix$ (Fig. 56) where $N$ is the number of the input parameters considered in the testing process and $M$ is the product of the number of elements of the inputs (the number of selected test data values). Here it is important to highlight that whereas in this case study the algorithm used to generate the test sequences is an all combinations one, other algorithms can be used (such as pair-wise or any n-at-a-time combinations) according to the specifics of the testing process at hand.

The next step after generating the $TestDataMatrix$, involves creating a test signal (as per the function specified during the formalization of partitions)
and data selection of the FRTP), which captures how these data values evolve over time.

The test signal can be visualised as a two dimensional matrix where the first column describes the time aspect of the signals while the second column captures how the test data values evolve over time. The time dimension of the test signal in this case study, is built by simulating a linear time, which progresses at a constant speed (in other words the simulated seconds all share the same inherited length/time-span) and the signal between each two test data values is built by interpolation.

The approach on how to generate the test signal is dependent on the environment where it is developed, but in all programming languages the issue simply entails translating a few concepts from linear algebra into a computer program. The issues mainly revolve around block matrix manipulation and matrix transposition. Selections of code snippets programmed in MATLAB are given in Appendix D.

For the MC/DC aspect of the CTMCONTROL the tests can be generated via using an all combinations algorithm or an all permutations algorithm on the parameters of interest (Table 31).

Since test generation for the MC/DC via the all combinations algorithm was already used and discussed in the controlled experiment section of my thesis, this case study presents the all permutations algorithm. While making the case on why one algorithm is better and in which circumstances it is of benefit is outside the scope of this thesis, the rationale behind the development of this algorithm is outlined below.

One of the underlining issues when generating tests to exercise logical statements, is the masking effect that logical operators can have.

To clarify this issue, let us consider a state transition guard $G$, where: $G = A \text{AND} B$. If $A$ evaluates to $false$, $B$ will not be executed. More generally speaking, a $false$ statement will mask all the other statements, which are logically connected to it via an ‘AND’ operator. Similarly, a $true$ statement
will mask all other inputs, which are logically connected to it via an ‘OR’ operator. The test cases whose outputs are not observable, typically do not count in terms of what MC/DC coverage can be credited. What is more, the fact that due to this ‘masking’, part of the logical statements are simply not exercised by tests, creates the danger of having control aspects of the SUT, which are completely untested. What happens when that part of the logic is eventually executed, possibly when the SUT is in use?

One very public example that comes to mind is Arianne 5 (Lions et al. 1996) where a simple unhandled floating point exception triggered a chain of events that caused the erroneous self-destruct of the launcher in its maiden flight.

The process of guaranteeing a 100% MC/DC coverage in all possible combinations of the different logical operators in a transition guard, in some cases, is also linked with being able to determine the transition or the state a Statechart is at any given time. While this problem has been widely researched (as discussed in Chapter 3 of this thesis) the existing solutions usually involve procedures that are very time expensive and impose very restrictive requirements on the Statecharts. Restrictions that often make the use of these algorithms unfeasible for real-world industrial systems.

Generating tests for all the permutations of A, B, and C being toggled between true and false, cannot guarantee a 100% executability and/or observability of all the test cases for all types of complex logical structures. The approach though can increase the hit rate of executable test sequences due to the very nature of permutations. For example in a statement like: S= A AND B OR C, an all permutations algorithm, there will be a test sequence in which when A is true, B is also true making so sure that B is executed. In this context, an all permutations algorithm can be seen as a pragmatic approach to mitigate the inherent executability and observability issues of this category of tests.
5.2.4. The Coverage Reports

The system was tested via the CTM classic mode, the MC/DC mode of the CTM\textsubscript{CONTROL}, which generates only the MC/DC coverage tests (via the all permutations algorithm), and via the full CTM\textsubscript{CONTROL} mode. For testing the SUT in the CTM mode were generated 7291 tests, for testing it in the MC/DC mode 367 tests, while the full CTM\textsubscript{CONTROL} mode generated 7658 test.

The MC/DC coverage achieved via the classic CTM approach is depicted in Figure 57, Figure 58 depicts the MC/DC coverage achieved via the MC/DC aspect of the CTM\textsubscript{CONTROL}, and Figure 59 depicts the MC/DC coverage achieved via the full CTM\textsubscript{CONTROL}.

<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Hierarchy/Complexity:</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1. TestHarness\WithAllFinalFinalFare</td>
</tr>
<tr>
<td>2. ... ACCSubsystem</td>
</tr>
<tr>
<td>3. ... ACCCntrl</td>
</tr>
<tr>
<td>4. ... ... SF, ACCCntrl</td>
</tr>
</tbody>
</table>

**Figure 57: CTM classic**

<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Hierarchy/Complexity:</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1. TestHarness\WithAllFinalFinalFare</td>
</tr>
<tr>
<td>2. ... ACCSubsystem</td>
</tr>
<tr>
<td>3. ... ACCCntrl</td>
</tr>
<tr>
<td>4. ... ... SF, ACCCntrl</td>
</tr>
</tbody>
</table>

**Figure 58: The MC/DC only coverage with all permutations**
The coverage reports show that there is a 14% improvement in the MC/DC coverage achieved via the CTM\textsubscript{CONTROL}. In this case study not all the SUT parameters were considered and furthermore a choice was made that for the MC/DC testing aspect to only include those parameters, which one could reasonably expect to have visibility from the specification level. Again the testing used in this case study was pitched somewhere mid-process, i.e. considering an incremental approach to testing where different parts of the system are tested incrementally where other parameters are added as the development process progresses.

While the controlled experiment showed an unobfuscated result, the case study has been developed to mirror more closely a real-life project where the SUT is developed in cycles. It reflects the fact that different points in the testing process the achieved structural coverage can be restricted or obfuscated by factors such as a partial implementation of SUT functionality and a restricted visibility a tester can have on some of the control logic of the SUT implementation.

### 5.3. Conclusions

In this Chapter, we have presented a validation of the proposed CTM\textsubscript{CONTROL} via a controlled experiment approach. CTM\textsubscript{CONTROL} delivered a 67% improvement in MC/DC coverage in a controlled experiment and a 14% improvement in an automotive case study. The results point to the fact that CTM\textsubscript{CONTROL} can capture a group of errors in the control-logic of the SUT, which would not be captured via a classic CTM approach.

The results from the controlled experiment and the case study are analysed in the next chapter as are all the contributions of this thesis.
6. Analysis of Research Contributions

“If you are going to teach people to make observations, you should show that something wonderful can come from them. I learned then what science was about: it was patience. If you looked, and you watched, and you paid attention, you got a great reward from it — although possibly not every time.”

Richard Feynman, 1969

This chapter analyses the results of this research. It does so by analysing the results of the controlled experiment and of the case study; by reflecting on the answers for the research questions and evaluating the overall contributions of the research.

6.1. The Approach Followed for the Analysis

The analysis is conducted through the lenses of the research questions and of the evaluation criteria discussed in the first chapter of this thesis. First, the rationale behind CTM\textsubscript{CONTROL} is analysed from a theoretical point of view. The analysis continues with the scrutiny of the results observed in the controlled experiment and in the case study. Following that, the proposed FRTP is brought under focus.

The discussion then continues into investigating whether the research questions posed for this research were answered and in elucidating the respective answers. Finally the research contributions are analysed through the lens of the evaluation criteria.

\footnote{Later in the same speech quoted in the previous chapter, Feynman continues to address the question “What is science?” (Speech given at the National Science Teachers Association in New York City and printed in The Physics Teacher, (7), 6, 1969, pp. 313-320)}
A conceptual view of the structure of the logic followed in this analysis is depicted in Figure 60.

![Conceptual view of the structural logic of the analysis](image)

**6.2. Analysing the Results**

*Theoretical reflections*

The proposed CTM\textsubscript{CONTROL}, as discussed in Chapter 4 of this thesis, can detect a group of errors, which would not normally be caught via the classic CTM. CTM\textsubscript{CONTROL} allows testing the dependencies among the input values and the control-logic of the SUT.

Reiterating in a summary form here, for a state-transition guard $L = A \land B \lor \neg C$, where $A$, $B$ and $C$ are logical predicates, the classic CTM would not necessarily target the ‘$\neg C$’ predicate. Scenarios such as ‘$\neg C$’ are important to test both for the presence of the required functionality, and, the absence
of unwanted functionality. Outcomes such as undesired activations could easily hide behind this type of logical control structures. The case of the Boeing crash in 1999 (mentioned in Section 4.2), or the maiden flight of Ariane 5 are just two examples of very public incidents brought on by undesired activations. The tests generated for the MC/DC aspect of CTM\textsubscript{CONTROL}, can be targeted to test this type of logical constructs, which could cause unwarranted activations.

To focus this analysis into the automotive software domain, let us\textsuperscript{46} consider a software function, which instructs an airbag to be deployed. The function makes the decision based on the inputs received by a number of sensors where each of them, by itself, could cause the deployment of the airbag. In other words, a malfunction of just one of the sensors can cause an undesired activation of the airbag. While an airbag’s erroneous deployment would always be problematic, it would be particularly so if the airbag caused an injury. Hence, let us assume that after a hazard and risk analysis it is decided that the scenario where each of the sensors can independently cause an erroneous deployment of the airbag, would present a real danger only when the car is travelling at speeds higher than a specific threshold or if the seat was occupied by a child. This control logic can be written as:

\textbf{Equation 3}

\[(A \lor B \lor C) \land V < \text{Threshold\_Vel} \land \text{PassengerIsAChild} = \text{FALSE} \Rightarrow \text{AirbagDeployed}.\]

Where, A, B and C represent the respective input values sent from the sensors a, b and c.

Tests generated via a CTM approach would not intentionally cover all the possible combinations of the \textit{true} and \textit{false} inputs being OR-ed, AND-ed and NOT-ed. While CTM\textsubscript{CONTROL} can generate tests to exercise the separate effect of each of the sensor inputs (to deploy or not to deploy), of the car velocity being below or above the given threshold and of the passenger

\textsuperscript{46} The plural in this section is used to denote (inclusively) the author of the thesis and the reader.
being a child or an adult. Indeed, CTM\textsubscript{CONTROL} if targeted at exactly this specific control-logic level, can target all the different MC/DC permutations of the atomic logical predicates toggled between \textit{true} and \textit{false}.

This brings me to the next point. CTM\textsubscript{CONTROL} can be pitched at different ‘test colour’ levels. While its primary focus could be a black or a nearly black-box approach, as pointed out in Chapter 4 and reproduced for ease of reading here in \textit{Figure 61}, CTM\textsubscript{CONTROL} can be used in the light-grey and white sections of the testing colour spectrum.

![Figure 61: CTM\textsubscript{CONTROL} mapped](image)

To fully clarify the importance of this point, I need to discuss one dynamic, which has served as one of the triggers for my original idea behind CTM\textsubscript{CONTROL}.

ISO 26262 imposes metric driven testing requirements such as MC/DC for the most critical automotive software. The requirements brought on by ISO 26262, while finding praise from a quality assurance point of view, have also created apprehension among the practitioners. This due to the costs involved in developing software that conforms to the higher ASILs. One of the mechanisms, which were adopted to mitigate costs has to do with the manner in which ASILs are calculated for systems, which incorporate a number of functions.
An ASIL level is not calculated for a physical system component but it is calculated for a function. That said, the ASIL associated with a function is inherited by the software and hardware elements that realize that function. The rule according to ISO 26262 (ISO 2011) is that the system will inherit the highest ASIL of its components. This translates into higher costs for producing a system, which incorporates even a single high ASIL component. To mitigate this effect, ISO 26262 (ISO 2011) has adopted a mechanism called ASIL decomposition. ASIL decomposition refers to a procedure, where the system can be split up so only the really critical part is developed according to the highest ASIL standard, and in the process reduce the QA metrics of the full system. This issue deserves a lot of interest on its own but the interesting aspect for my research is the fact that the MC/DC aspect of CTM\textsubscript{CONTROL} could be targeted at different test colour levels while testing the same SUT. It could be targeted at a coarser granularity level (for example system-specification level) for the functions that after decomposition fall under a lower ASIL and at a finer granularity level (for example at white-box level) for those functions that retain the higher ASIL requirements.

Recapitulating, CTM\textsubscript{CONTROL} has been built by extending an existing method so as not to introduce a steep learning curve for automotive software testers. When compared to the method, which it improves upon, CTM\textsubscript{CONTROL} can detect a new additional group of errors. The approach, in addition to the benefits of the classic CTM, also fulfills the MC/DC objective, and has been prototypically realized in the M/S/S environment, which is popular in the automotive industry.

\textit{Reflections on the observed experimental results}

As described in Chapter 5, CTM\textsubscript{CONTROL} is evaluated via a controlled experiment and an automotive case study and research on it has been published.

The SUT in the controlled experiment is a conceptualized model, which is abstracted from a simple automotive controller. The experiment is
constructed to isolate for observation a single state-transition of a state-based system. In the experiment, the MC/DC-only aspect of CTM\textsubscript{CONTROL} has full visibility of all the parameters involved in the logic of the state-transition guard. During the entire testing process, in both the CTM and the MC/DC-only approach, the SUT is forced to be in the transition of interest. One potential downside of this choice is that it will mask out any differences that the two approaches deliver in terms of overall model coverage. That said, since the full CTM\textsubscript{CONTROL} cumulates the model coverage capabilities of CTM and those of the MC/DC-only approach, the need to ‘look under that mask’ is not of a crucial importance.

During the MC/DC-only testing of the SUT, a 100% MC/DC coverage was achieved. At the same time, a classic CTM approach achieved a 33 % MC/DC coverage. From the 33% coverage of CTM, it can be safely deduced that there is a group of errors, nominally a subset of all those errors for which the MC/DC concept was initially proposed, which the CTM will not detect. Conversely, the high MC/DC coverage achieved via the MC/DC-only approach points to the fact that, that same group of errors can be caught by the proposed CTM\textsubscript{CONTROL}.

The automotive case study presents a prototypical realisation of CTM\textsubscript{CONTROL} used to test a SUT, which is studied and referred to in literature by a number of researchers. The rationale behind choosing as a SUT a well-known case-study such as the ACC, is to facilitate reproducibility of the results reported here and hopefully to facilitate future research that improves on the solution presented here.

The testing in the case study was designed to include only a subset of the SUT parameters. This aimed to simulate a lack of full visibility of the SUT parameters and/or testing performed somewhere mid-process in an approach where the functionality to be tested is built incrementally.

The test harness for the ACC was executed in CTM mode, MC/DC aspect of the CTM\textsubscript{CONTROL} mode and in full CTM\textsubscript{CONTROL} mode (Table 32). The
classic CTM approach generated 7291 tests and achieved a 21 % MC/DC coverage. The MC/DC-only mode delivered a 12% improvement on the MC/DC coverage with only 367 tests. Finally, the CTM<sub>CONTROL</sub> mode achieved a 35% MC/DC coverage, which translates into a 14% improvement compared with the classic CTM.

<table>
<thead>
<tr>
<th></th>
<th>No of the considered input parameters</th>
<th>Number of tests</th>
<th>Time of test execution [simulated seconds]</th>
<th>MC/DC coverage [%]</th>
<th>Change of MC/DC coverage as compared with CTM [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTM</td>
<td>5</td>
<td>7291</td>
<td>7291</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>MC/DC-only</td>
<td>3</td>
<td>367</td>
<td>367</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>CTM&lt;sub&gt;CONTROL&lt;/sub&gt;</td>
<td>5 (3 from them were considered also for the MC/DC aspect)</td>
<td>7658</td>
<td>7658</td>
<td>35</td>
<td>14</td>
</tr>
</tbody>
</table>

The automotive case study concurs with the controlled experiment in pointing to the fact that CTM<sub>CONTROL</sub> delivers a considerable improvement in terms of MC/DC coverage. In the case study, improvement was observed even when including only a subset of the SUT parameters and the coverage benefits were delivered with a relatively small number of additional tests. Here it is important to highlight the fact that the number of the additional tests generated by the CTM<sub>CONTROL</sub> depends on the number of the parameters involved on the control logic of the SUT. Hence, the fact that CTM<sub>CONTROL</sub> only requires a small number of additional tests is not a generalizable conclusion.

The MC/DC concept translated into a tree structure also facilitates the use of other existing classification tree tools. These tools can be used as they are or perhaps might require the development of an additional algorithm for the
combination of classifications (i.e. for the test generation). CTM\textsubscript{CONTROL} per se allows the tester to enter the values of interest via a Graphical User Interface or the MATLAB command line. The tests are generated automatically using an algorithm written in MATLAB.

**Reflections on the Formalization of a Reactive Testing Process**

QA standards such as ISO 26262 (ISO 2011) require formality to be incorporated in the testing process, but, give scant guidance on which formality to use and on the methodology of how to use it. There are vast differences in the precision and mathematical rigor of methods and processes, which are called ‘formal’. Even so, in all its varying degrees, formality can be perceived by industry as a very expensive activity, which requires a high degree of mathematical aptitude and knowledge, and which is perhaps to be used as little as possible.

The proposed FRTP appeases the QA standards’ requirements for formality and breaks down the process of introducing formality into relatively simple steps, allowing some flexibility for the tester.

A full validation of the FRTP is outside the scope of this research. That said the test harness for the ACC was developed following all the steps of the proposed FRTP, hence in the process, illustrating its use.

Reportedly (Hinchey 2003, Parnas 2010), one of the underlying causes of a low uptake of formality by industry is an inadequate education of practitioners. Even in the cases when formal methods are taught as part of software engineering courses, often the scaling up of this knowledge into a real industrial example can be perceived as a too difficult task. The FRTP uses Z notation to capture the specification, test objectives, test reactivity and the partition heuristics used for test generation. Z was chosen as one of the notations, which are often taught in third level education. When need arises and circumstances such as finances and available skills permit, the aspects captured in Z notation can be used to formally verify or prove key
aspects of the model\textsuperscript{47} (Hinchey 2003). Most importantly these steps can be introduced as needed instead of abiding by an all or nothing mindset. In all circumstances, an important benefit they bring is that of clarity of specification, tasks and objectives.

The FRTP suggests a workflow with a view to help at a methodological level. It defines ordered steps and a flow of logic on how to move between them (Bowen and Hinchey 2006). The proposed workflow per se has been informed and abstracted by reports in literature about the state of practice.

The Taxonomy of Reactive Testing, which is a component of the FRTP, helps the tester to elucidate the different reactivity dimensions which can be incorporated in a testing process. Z notation is used alongside the M/S/S environment in an automotive case study in order to show how a formal reactive testing process can be split in small and concrete steps all the while by keeping the implementation in an environment, which is familiar for industry. The Z schemas provided in Chapter 4, which illustrate different types of test reactivity (as part of the TRT), target specific automotive software issues with a view to aid understanding and uptake of the approach by industry.

Overall, during the course of this research, the Z notation was found to be more than satisfactory in terms of the syntax and structures needed to document the different facets of an automotive testing process.

Where does CTM\textsubscript{CONTROL} fit into the MBT Taxonomy?

The capabilities of CTM\textsubscript{CONTROL} have been mapped (Fig. 62) into the revised MBT taxonomy proposed in Chapter 4. A red dot near a branch of the MBT denotes a dimension of the MBT, which applies to CTM\textsubscript{CONTROL}.

\textsuperscript{47} Outside the scope of this research.
In terms of structural model coverage, depicted in Figure 62, $\text{CTM}_{\text{CONTROL}}$ delivers MC/DC coverage. In terms of data coverage capabilities, it inherits those of CTM.

$\text{CTM}_{\text{CONTROL}}$ can deliver requirements coverage when the requirements are translated in terms of parameter values or control logic. When $\text{CTM}_{\text{CONTROL}}$ is used via typical automotive toolsets such as M/S/S, the tools can be used to connect the feature covered by testing to the actual requirement\(^{48}\).

Test case specifications are comfortably supported within $\text{CTM}_{\text{CONTROL}}$ when they can be translated into a classification tree. When this is not possible, $\text{CTM}_{\text{CONTROL}}$ can use the capabilities of a full industrial environment such as M/S/S. In these cases, the test data can be generated offline and executed together with the rest of the test sequences, or can be modelled directly into M/S/S and again executed as part of the other test sequences. The same rationale is valid for the generation of fault-based tests.

As seen in the case study, the test-selection criteria can be based on the required reactive behaviour and the decision is made on-the-fly. The tests are designed manually but generated automatically. Test generation is done offline while test execution (including test control) is done online. The test generation is done offline.

\(^{48}\text{This part has not been implemented in the prototype but the functionality already exists in the M/S/S environment or when M/S/S is connected to DOORS. The later being a popular setup in the automotive industry.}\)
execution is done via a Model-in-the-Loop approach and tests can be executed reactively.

6.3. The Answers to the Research Questions

The analysis performed in the previous section of this chapter, facilitates the task of investigating whether the research questions posed in Chapter 1 were answered and the manner in which they were answered. For this, I will revisit the research questions one at a time.

**Research Question 1:** Can a software testing method, which is in common use in industry be improved in such a way that brings that method closer to the latest QA requirements?

The relevant QA standards regulating automotive software require the incorporation of formality in the testing process and require MC/DC coverage for its most critical software. This research identified CTM as a testing method, which is widely used in industry.

The proposed CTM\text{\textregistered}\text{\textcopyright}, incorporates MC/DC capabilities into the classic CTM. The proposed FRTP illustrates a way to incorporate formality throughout the testing process.

**Research Question 2:** Can this proposed method be implemented via toolsets already popular in the automotive industry?

There can be a gap between the methods and approaches proposed by academic researchers and those adapted by the industry. This was the rationale behind aiming to use only toolsets already in wide use in industry rather than build new tools or develop new notations. The proposed CTM\text{\textregistered}\text{\textcopyright} was developed in the M/S/S environment where a reported 50% of the functional behaviour for automotive embedded systems is modelled (Helmerich et al. 2005). Furthermore, since the MC/DC aspect of CTM\text{\textregistered}\text{\textcopyright} has been translated into a tree structure, the method can also be implemented in other toolsets already in use for classification tree based
testing and/or other toolsets that can be connected to the M/S/S environment.

**Research Question 3:** *Can the improved method be used within a software testing process, which incorporates formality?*

As illustrated via the case study and analysed in the previous section, CTM\textsubscript{CONTROL} can be used within a formal reactive testing process.

**Research Question 4:** *Finally, how can the worthiness of the proposed method be evaluated?*

The proposed method was evaluated through a controlled experiment and an automotive case study. The observation of the experimental results demonstrated that the new method can target a class of errors, which would not necessarily be caught by the classic CTM.

The improvement in structural coverage delivered by the proposed method was measured quantitatively.
6.4. Considerations Based on the Evaluation Criteria

The evaluation criteria for the research contributions which have been informed by literature and initially presented in the “Introduction” chapter of my thesis, will be revisited here to evaluate this research’s contributions.

Evaluation Criterion 1: The proposed testing method should target a specific requirement or a number of requirements posed by QA standards (ISO 2011, IEC 1998).

As previously discussed, the ISO 26262 standard has specific recommendations for software testing, including the minimum testing requirements for specific ASILs and a number of recommended test approaches such as requirement-based tests, equivalence-classes-based tests and structural coverage tests.

As part of its recommendations for software testing, ISO26262 recommends that formal notations should be used for the higher ASILs throughout the development process including specification, design, architecture and testing. For example Figure 63 depicts the standard’s recommendations for the methods to be used for the verification of requirements for the different ASILs.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal verification by walkthrough</td>
<td>++</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Informal verification by inspection</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Semi-formal verification</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Formal verification</td>
<td>0</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

"0" Neither recommended nor not recommended for this ASIL.
"+" Recommended for this ASIL
"++" Highly recommended for this ASIL

Figure 63: Recommended methods for verification of requirements by ISO 26262

The formal verification recommended in the standard is defined as an activity, which is used to ensure the correctness of the system under test as
defined in the formal specification of its behavior. In other words, it does not necessarily enforce the same mathematical process, which is typically understood as formal verification by formal methodists. It can be rather construed as a recommendation to formally specify the requirements, which can serve as the basis for any potential verification activities.

The ISO standard also introduces the use of specific coverage metrics for certain ASILs (as depicted in Figure 64). More specifically, MC/DC structural coverage is recommended for ASILs A, B and C and highly recommended for ASIL D. The rationale behind the requirement of specific coverage requirements is linked to the idea of completeness of test cases and avoidance of unintended functionality.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Statement coverage</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>MC/DC coverage</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>

"++" The method is recommended for this ASIL
"+++" The method is highly recommended for this ASIL

Figure 64: Structural coverage metrics recommended by ISO 26262 Part6

As highlighted in the previous sections of this analysis, the proposed method delivers the recommended MC/DC coverage capabilities and the proposed FRTP provides the methodological pointers on how to incorporate the recommended formality in the testing process.

**Evaluation Criterion 2**: The improvement of the proposed method over an existing method, should be measurable, in order to facilitate quantitative measurement in an experimental setup.

The improved structural coverage delivered by CTM\textsubscript{CONTROL}, by its own nature, is quantifiable. In the controlled experiment and the case study the improvement delivered by the proposed method was clearly measurable. Furthermore, at a more abstract level of measurement, CTM\textsubscript{CONTROL} was shown to capture a group of errors not previously covered by CTM.
Evaluation Criterion 3: In order not to introduce a steep learning curve, which could serve as a barrier for its uptake by industry, the method should make possible the use of toolsets that are already used in industry (deduced from research reported by (Hinchey 2003, Broy 2009, Parnas 2010)).

Unfamiliarity with formal tools developed in a pure academic environment can pose a perceived barrier for industry uptake. Hence the prototypical realisation of CTM_{CONTROL} is implemented in the M/S/S environment where reportedly 50% of the functional behaviour for automotive embedded systems is modelled (Helmerich et al. 2005). The proposed formality is split in small and graded steps, which can be used as needed.

Evaluation Criterion 4: The method should incorporate graphical modelling for software in order to facilitate communication and understanding as early as possible in the development process. This is seen as especially important in the context of the multidisciplinary nature of automotive software development, which typically incorporates aspects from mechanical, electrical and software engineering disciplines (Broy 2009).

While in terms of formality the FRTP is underpinned by Z notation, the prototypical implementation of the method is executed in the M/S/S environment where a more light-weight formal notation such as Statecharts is used. The graphical aspect of both Simulink as an environment, the Statecharts and the graphical functions incorporated into the implementation, all fulfil the criterion of incorporating graphical solutions in the approach.

Evaluation Criterion 5: The method should allow the benefits of formal notation, methods and processes to be harnessed (typically to cater for safety-critical systems or safety-critical functions in a system). The incorporation of formality should be flexible and introduced as needed. It should be ‘tailorable’ to the needs of the company or the project at hand.

As already discussed in the previous sections, the Formalization of a Reactive Testing Process was proposed and the proposed CTM\text{CONTROL} was used within this formal approach. Again, the proposed formality can be introduced as needed as per the methodological pointers provided. Alternative ways of capturing requirements unambiguously such as the tabular approach used for the formalisation of data partitions were also demonstrated.


The proposed approach was implemented in a prototypical fashion in a testing harness, which includes reactive testing. The reactive scenarios are implemented in Simulink. Furthermore, in Chapter 4 of this thesis a reactivity taxonomy alongside Z schemas illustrating each of its branches were provided. The validation of the proposed reactivity taxonomy is outside the scope of this research. The proposed taxonomy is meant to be used as an evolving artefact, where the tester can choose from the depicted reactivity dimensions or add new ones. This, since test reactivity often can be understood too narrowly as a concept. The proposed taxonomy, albeit an evolving artefact, has the potential to help with the elucidation of the different dimensions of reactivity, which are necessary for an effective testing process. The order brought to information by the taxonomy can enhance the tester’s ability to elucidate all the possible type of reactivity patterns that can be used in the process at hand. The taxonomy can also be used to provide structure to further work, be it of automation or implementation of meta-models for the types of reactivity.
6.5. Conclusions

In this Chapter, we have presented an analysis of the research contributions reported in this thesis.

This research has resulted in clear answers for the questions that motivated this research. Furthermore, the research contributions fulfil the evaluation criteria, which were identified in literature as important qualitative indicators within the scope of this research.

The next chapter discusses some of the main conclusions as well as the limitations of the proposed solutions and possible directions for future work.
7. Conclusions, Limitations and Future Work

“Si ventus non est, remiga.”

Latin proverb.

This chapter starts by showing a snapshot of the background of this research. It highlights links between some of the key ideas, which emerged from the review of the state of art. The chapter follows with a synthesis of the contributions of this research, the limitations and future work. This chapter concludes with a short discussion of some current trends, which are expected to have an impact on the field of automotive software.

7.1. Research Background

This research set out to contribute to the body of knowledge in the field of model-based testing of safety-related software with a particular interest in the embedded automotive software.

A review of literature highlighted a number of competing forces, which continuously shape the field of automotive software testing. The focus of this research was drawn into the complex interplay between the need for better testing methods, the relevant standards’ (ISO 2011, IEC 1998) QA requirements on formality and testing metrics, and the industry’s need to produce standard-compliant software in a practical and financially feasible way (Fig. 65).

---

49 If there is no wind, row.
Current moves towards standardization such as AUTOSAR (AUTOSAR 2006) and the QA requirements of the standards that regulate automotive software such as ISO 26262 (ISO 2011) increase the need for reliable and confidence inspiring testing approaches.

Testing automotive software is a very complex and multifaceted field. This research’s interest was captured by the standards’ requirements for specific test metrics and formality to be incorporated in the software’s development lifecycle.

First of all, ISO26262’s (ISO 2011) requirement for MC/DC coverage captured this research’s interest for a number of reasons. The most
prominent reason being that one of the strongest arguments, typically presented by the automotive industry, to overcome reliability assurances and the legal issues standing in the way of widely deployed autonomous and X-by-Wire functionality in vehicles, is the avionics software. The evidence presented to support this argument is the fact that some autonomous features and X-by-Wire technologies\(^{50}\) have been used successfully by avionics for a number of years. Any move towards X-by-Wire or (in some capacity) autonomous vehicles, requires that at least the same level of quality assurance as that required for avionics software has been adhered to. MC/DC coverage is required by avionics QA standards (such as DO-178C/ED-12C) for all Level A (most critical) software for airborne systems before they can get an official certification. Hence, the MC/DC coverage proposed by the emerging ISO26262 (ISO 2011) standard has a heightened importance.

ISO26262’s (ISO 2011) requirement for MC/DC coverage was met with a level of apprehension by a number of industrial stakeholders. This was due to the costs involved in achieving MC/DC coverage for a full system in order to appease QA requirements for that system’s safety-critical functions. This was partially addressed via the concept of ASIL decomposition (see Section 6.2 for further detail). Even so, ISO26262 (ISO 2011) provides very scant methodological guidance on how to achieve MC/DC and there still exists a level of confusion on how exactly to implement it.

Regarding the QA requirements for formality, while there are a significant number of impressive formal testing approaches, they are often proposed for idealized systems and can prove too restrictive or challenging for industrial use.

The standards, while recommending formality, confine themselves to mentioning a wide spectrum of techniques and do not elaborate on the specific criteria that would qualify a method or notation to reach the desired

\(^{50}\) Such as Steer or Brake-by-Wire.
‘formality benchmark’. Neither do they incorporate any type of methodological pointers on how to incorporate formality in the testing process.

This research analyzed the specific type of formality used in a selection of twenty toolsets, which are targeted at state-based systems.

Along with other conclusions (reported in Section 2.9), it emerged that only 20% of the reviewed tools are commonly used both in industry and academia. Graphical approaches appear to be more popular and, in this context, Statecharts appear to be the most popular approach in tools, which are used by both academia and industry.

This research also analysed the testing algorithms, which underpin a number of tools that aim to automatically generate tests for state-based systems.

Overall, the major issues faced by formal testing are: the combinatorial explosion of the solution; and, the added complexity when dealing with hierarchy, concurrency and communicating states, which share data interactions. Furthermore, most of the proposed strategies and solutions in this field have the common limitation of being based on a number of assumptions and preconditions on the SUT, which can prove too restrictive or even unrealistic for real world testing. Additionally, the unfamiliarity of practitioners with tools proposed by researchers, built in an academic setup and catering for idealised conditions, creates a real or perceived barrier for uptake by industry (Dill and Rushby 1996, Holloway and Butler 1996, Broy 2009).

Overall, achieving a completely automatic on-the-fly test generation remains an eluding goal as yet to be achieved and the automotive industry is hungry for standard-compliant testing methods targeted at the embedded automotive software.
7.2. A Synthesis of the Main Contributions

This research gave clear answers to the research questions initially posed in Chapter 1 and analysed in Chapter 6 of this thesis. The contributions reported in this thesis fulfil the evaluation criteria which were identified as important qualitative indicators for the scope of this research (Chapter 1).

The thesis proposes a new testing method called CTM$_{CONTROL}$, which improves upon the existing CTM method by bringing it closer to the QA requirements of the latest standards, which regulate the automotive software field.

Informed by the industry’s reluctance into adopting ‘yet-another-method’, the research focused on proposing a practical solution, which makes use of methods and toolsets that are already familiar to the automotive software tester. Crucially, CTM$_{CONTROL}$ has been built by extending an existing method with the intention of not introducing a steep learning curve for testers.

The proposed method incorporates MC/DC coverage and can find a group of errors, which were not targeted by the method it improves upon. The additional group of errors captured by CTM$_{CONTROL}$ includes errors, which can cause unwanted activations or unwanted feature interactions in a system.

In CTM$_{CONTROL}$, the tests are designed manually and generated automatically. Test generation is done offline while test execution (including test control) is done online. Test selection can be done reactively on-the-fly and the tests can be executed reactively via a Model-in-the-Loop approach. The Model-in-the-Loop approach here refers to the fact that both the system under test and the environment with which it interacts are simulated as models.

CTM$_{CONTROL}$ was prototypically implemented in the M/S/S environment, which is very popular in the automotive industry. The method was evaluated experimentally via a controlled experiment and a case study, which
incorporates an experimental logic.

In response to the QA requirements for the incorporation of formality in the testing process (ISO 2011), this thesis also proposes the Formalization of a Reactive Testing Process (FRTP). The FRTP builds a pathway which connects reactive testing to the field of formal testing (Fig. 66).

![Figure 66: The CTM\textsubscript{CONTROL} and FRTP connection](image)

This ‘connection’ facilitates the incorporation of formality in the reactive testing process. The FRTP aims to facilitate the incorporation of formality in industrial practice and it has been built to support the needs of automotive safety-related software testing.

The use of CTM\textsubscript{CONTROL} was illustrated within the FRTP. When used outside FRTP, CTM\textsubscript{CONTROL} delivers the MC/DC criterion. Whereas when used within the proposed FRTP, CTM\textsubscript{CONTROL} is mapped into the field of formal testing and in addition to the MC/DC criterion it delivers the incorporation of formality in a reactive testing process (Fig. 67).
As part of the FRTP, this research also proposes changes on a model-based testing taxonomy proposed by (Utting et al. 2006) and later enhanced by (Zander-Nowicka 2009) and a new Taxonomy for Reactive Testing (TRT). The change in the model-based testing taxonomy consists of including test reactivity in the test–selection criteria to account for the cases when test generation is performed on the fly and reactive test behaviour decides the trajectory among all of the possible test paths.

The TRT brings a contribution in the field of reactive testing and when used within FRTP, it facilitates the formalization of test reactivity.

In order to aid the use of formalization in industry, examples in Z notation are given for each of the dimensions of the TRT. The examples are meant to serve as informational models, which can guide the ‘typical automotive software tester’ in the process of formalizing test reactivity.
7.3. Limitations and Future Work

This research has a lot of potential to be developed further in a number of directions.

The ISO26262 standard (ISO 2011), which regulates automotive software, has not formally defined the MC/DC criterion and it has not adopted any of the formalizations proposed by other researchers (Vilkomir and Bowen 2006, Kirner 2009, Comar et al. 2010). Since this research is focused on standard-compliant testing, in this thesis, MC/DC coverage is intentionally left not (formally) defined. That said a clear reference is made to the formalization provided by Vilkomir and Bowen (2006) as a formalization which covers the MC/DC criterion used in this thesis.

The adoption of a formally defined MC/DC criterion by ISO 26262 has the potential to create interest in a number of issues. For example a full formal definition of the test-generation algorithms to use for inputs connected through different logical operators (AND, OR, NOT, XOR) and their different combinations is a very interesting area to be addressed in future work.

Future work includes the implementation of a tool which can automatically generate a minimal test-set to achieve full MC/DC coverage for any combination of logical structures. This functionality could be developed via implementing a constraint solver, which takes in consideration the specificities of the different logical operands in terms of achieving MC/DC coverage. The algorithm could also be refined into highlighting the tests, which are needed for a full MC/DC coverage but are not executable. Some preliminary steps have been taken to liaise with other researchers that are working in this area.

Another interesting direction for further work would be to expand on the synergy of CTM\textsubscript{CONTROL}, the FRTP and the M/S/S environment. For instance, Simulink blocks could be developed so as to be used as patterns, which can help to quickly include the desired test reactivity or the desired
test metrics. The envisaged Simulink blocks could then be initialised or refined according the needs of the tester and help her to achieve the desired results faster.

Especially since the M/S/S environment is deeply rooted in mathematics, it would be of interest to investigate the addition of a strict formal tool to it. An add-on tool, which could make use of a strict formal notation would be a very good match for the M/S/S environment. For example this tool could automatically make the link between specifications written in a strict formal notation, and the system modelled into Simulink, Stateflow etc. This would enforce the feasibility of formal verification, and in more general terms mathematically prove the correctness of critical features. An initial contact has been made with researchers in Queen Mary University of London, which are working in a tool which connects simulations in the M/S/S environment with the PVS Specification and Verification System.

$\text{CTM}_{\text{CONTROL}}$ while it has been used for testing automotive systems, it has not been used in an industrial setup. Using it in an industrial setup, would allow to assess $\text{CTM}_{\text{CONTROL}}$ in terms of time and effort needed from the tester as well as to further evolve the taxonomy of test reactivity. Additionally, other issues such as test reusability and other metrics could be better investigated in an industrial setting.

A number of aspects reported upon in this research (such as the TRT and the FRTP) could benefit from a validation in an industrial setup.

While $\text{CTM}_{\text{CONTROL}}$ was implemented in the M/S/S environment so as to match the automotive industry’s preferences, it would be of interest to investigate the portability of the method into other environments.

Finally, on a personal note, I would really appreciate if the ideas proposed in this thesis would be further developed and improved upon by other researchers.
7.4. Closing Remarks

The automotive industry faces continuous cost pressure by QA requirements and regulation, increasing competition and a poorly performing economy.

Research and development spending in the automotive industry in 2005, was estimated at 68 billion Euro and was projected to rise to 800 billion Euro in 2015 (Oliver Wyman 2007).

The electronics (including software) are estimated to represent 40% (Scuro 2013) to 60% (Oliver Wyman 2007) of the innovations in vehicles. Currently, one of the strongest growing sectors in the automotive industry is fuel efficient technology, especially in the form of hybrid and electric vehicles. According to an Experian study, released on 22 April 2013, the market share of electric vehicles in the United States grew by 40.9% in a year; going from 2.2% in 2011 to 3.1% in 2012 (Experian Automotive 2013). In hybrid or electric vehicles, the value added to a state-of-the-art vehicle by electronics is estimated to be as high as 75% (Scuro 2013).

Areas such as active safety, seamless connectivity, infotainment and increasing comfort in the driving experience are all projected to grow.

In all these areas software plays a prominent role and as a result automotive software is expected to increase dramatically both in size and in the features it delivers.

This, while quality assurance measures are on the rise, due to standardization efforts such as AUTOSAR and other legislation and standards such as the emerging ISO26262 (ISO 2011).

This level of complexity will need a matched level of rigorous development and testing techniques. In order to remain competitive, automotive software companies will need to adopt software development and testing strategies, which comply with the QA standards and do so in a cost efficient way.
8. Glossary of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Anti-lock Brakes System</td>
</tr>
<tr>
<td>ACC</td>
<td>Adaptive Cruise Control</td>
</tr>
<tr>
<td>AGEDIS</td>
<td>Automated Generation and Execution of test suites for DIStributed component-based Software</td>
</tr>
<tr>
<td>AML</td>
<td>Agedis Modelling Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIL</td>
<td>Automotive Safety Integrity Level</td>
</tr>
<tr>
<td>ASM</td>
<td>Abstract State Machine</td>
</tr>
<tr>
<td>ASML</td>
<td>Abstract State Machine Language</td>
</tr>
<tr>
<td>ASN.1</td>
<td>Abstract Syntax Notation One</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Automotive Open System Architecture</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CEPS</td>
<td>Common Electronic Purse Specification</td>
</tr>
<tr>
<td>CLP</td>
<td>Constraint Logic Programming</td>
</tr>
<tr>
<td>CTM</td>
<td>Classification Tree Method</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>D</td>
<td>DOORS</td>
</tr>
<tr>
<td>E</td>
<td>Electrical and/or Electronic</td>
</tr>
<tr>
<td>ECM</td>
<td>Engine Control Modules</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EFSMS</td>
<td>Extended Finite State Machines</td>
</tr>
<tr>
<td>F</td>
<td>Formalization of a Reactive Testing Process</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>G</td>
<td>Graphical Format for TTCN</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>H</td>
<td>Hardware-in-the-Loop</td>
</tr>
<tr>
<td>I</td>
<td>Input / Output</td>
</tr>
<tr>
<td>I/O</td>
<td>Input Output Labelled Transition System</td>
</tr>
<tr>
<td>IOLTS</td>
<td>Input Output Symbolic Transition System</td>
</tr>
<tr>
<td>IOSTS</td>
<td>Implementation Under Test</td>
</tr>
<tr>
<td>L</td>
<td>Leiros Test Generator</td>
</tr>
<tr>
<td>LTS</td>
<td>Labelled Transition systems</td>
</tr>
</tbody>
</table>
### Glossary of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M/S/S</td>
<td>MATLAB/Simulink/Stateflow</td>
</tr>
<tr>
<td>MATLAB</td>
<td>MATrix LABoratory</td>
</tr>
<tr>
<td>MBT</td>
<td>Model-based testing</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Modified Condition/Decision Coverage</td>
</tr>
<tr>
<td>MiL</td>
<td>Model-in-the-Loop</td>
</tr>
<tr>
<td>MISRA</td>
<td>Motor Industry Research Association</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEMs</td>
<td>Original Equipment Manufacturers</td>
</tr>
<tr>
<td>OBC</td>
<td>Object Branch Coverage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFD</td>
<td>Possibility of Failure on Demand</td>
</tr>
<tr>
<td>PHACT</td>
<td>PHilips Automated Conformance Tester</td>
</tr>
<tr>
<td>PiL</td>
<td>Processor-in-the-Loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA</td>
<td>Quality Assurance</td>
</tr>
<tr>
<td>QM</td>
<td>Quality Management (System)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC/DC</td>
<td>Reinforced Condition / Decision Coverage</td>
</tr>
<tr>
<td>RTCA</td>
<td>Radio Technical Commission for Aeronautics</td>
</tr>
<tr>
<td>RTE</td>
<td>Runtime Environment</td>
</tr>
</tbody>
</table>
GLOSARY OF ACRONYMS AND ABBREVIATIONS

S
SBS   Sensotronic Brake Control
SCADE  Safety Critical Application Development Environment
SDL   Specification and Description Language
SIL   Safety Integrity Level
SiL   Software-in-the-Loop
STG   Symbolic Test Generator
SUT   System Under Test

T
TFT   Tabular Format for TTCN
TSC   Test Sequence Charts
TRT   Taxonomy of Reactive Testing
TT   Transition Tours
TTCN-3  Testing and Test Control Notation Version 3

U
U2TP   UML Testing Profile
UIO   Unique Input Output
UML   Unified Modelling Language

W
Wp    The partial W method

Numeric Literals
3GPP   Third Generation Partnership Program
9. References


CAST (2001) 'Rationale for Accepting Masking MC/DC in Certification Projects', available:


*Methods for Testing and Specification (MTS): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language* 2008a, ETSI.


REFERENCES


REFERENCES


REFERENCES


ITU 'Specification and Description Language - SDL', [online], available: http://www.itu.int/rec/T-REC-z [accessed 14 August 2013].


REFERENCES

MISRA Development Guidelines for Vehicle Based Software 1994, MISRA.
REFERENCES


REFERENCES


The Scilab Consortium 'Scilab', [online], available: [http://www.scilab.org/products/scilab](http://www.scilab.org/products/scilab) [accessed 03 September 2012].


REFERENCES


APPENDIX A – Partial Z Specification for an Adaptive Cruise Control

This appendix depicts a partial Z specification for the ACC and its test harness.

Partial natural language specification for the ACC

The ACC keeps the velocity of a vehicle to a value chosen by the driver while maintaining a safe distance from any physical obstacles in its path. In this case study, the ACC’s interface with the driver is a lever, which can be in one of five positions: Off, Neutral, Set+, Set- and Resume. On command from the driver, the ACC takes control of the velocity of the vehicle. The driver can set the required velocity via the Set+/Set- positions, which increment/decrement the velocity by a predetermined value (in this example by 5 km/h (1.4 m/s). The car also has radar technology and if an obstacle is detected in front of the car, it adjusts the speed so as to be at a safe distance from the obstacle in front. When the deceleration needed to avoid a detected obstacle goes beyond a safe value an auditory alarm warns the driver.

The ACC can be operated for any velocities up to 69.4 m/s and it can bring the vehicle to a full stop.

The driver must be able to increase the speed at any time by pressing the accelerator pedal, or reduce the speed by pressing the brake pedal. When the accelerator pedal is released, the ACC will regain control. Any time the brake pedal is pressed, the system must go inactive. Following this, when the brake is released and Set+, Set or Resume is pressed, the ACC becomes active and treats the vehicles’ current velocity as the selected speed. However, if an ‘Off’ has occurred in the intervening time, the Set+, Set and Resume commands do nothing.

The ACC’s interface with the driver

The Z notation used in this section follows the ISO/IEC 13568 standard (ISO/IEC 2002) and it has been type-checked via the CZT type-checker part of the Z Word tool (Hall 2005).

The ACC case study involves the use of real numbers, nevertheless due to the type-checker’s restrictions all the parameters are declared to be integers. To facilitate this, all the actual values within the Z schemata have been rounded up/down to their nearest integer value.
The ACC’s interface with the driver is a lever, which can be in one of five positions: Off, Neutral, Set+, Set- and Resume. To the lever positions it has been assigned a number: Off = 0, Neutral = 1, Set+ = 2, Set- = 3, Resume = 4.

\[ LEVERPOSITION = \{0,1,2,3,4\} \]

The Set+ and Set- change the speed of the car by 1.4 m/s (this, in this context, is called a step).

\[ SET\_PLUS\_MINUS::=SetPlus|SetMinus \]

\[
\begin{align*}
\text{SetVelocity} \\
\quad v,v': \text{VELOCITY} \\
\quad \text{Step: VELOCITY} \\
\quad \forall x: SET\_PLUS\_MINUS \Rightarrow x = \text{SetPlus} \Rightarrow v' = v + \text{Step} \\
\quad \forall y: SET\_PLUS\_MINUS \Rightarrow y = \text{SetMinus} \Rightarrow v' = v - \text{Step}
\end{align*}
\]

The highest velocity of the car is assumed to be 69.4 m/s (249.8 km/h).

\[
\forall v \in \text{VELOCITY} | 0 > v <= 69
\]

The driver can press the acceleration pedal, the brake pedal or not press any pedals. The system recognises this interaction via a series of pedal flags. Specifically, Pedal Flag = 0 means that no pedal has been pressed, 1 means the driver has pressed the acceleration pedal, while 2 means that the driver has pressed the brake’s pedal.

\[ PEDALFLAG = \{0,1,2\} \]

To facilitate type-checking, a Boolean type is also declared:

\[ BOOLEAN::=T|F \]

The Test Harness

The ACC model contains states, transitions, in-ports, out-ports

\[ STATUS::=State\_Off|State\_Standby|State\_CruiseByVelocity|State\_CruiseByDistance \]

\[
\begin{align*}
\text{State} \\
\quad \text{activeS: BOOLEAN} \\
\end{align*}
\]

\[
\begin{align*}
\text{InitState} \\
\quad \Delta \text{State} \\
\quad \text{activeS'}=F
\end{align*}
\]
Only one state or one transition of the ACC can be active at any one time.

\[
\text{Transition} \\
\text{activeT}: \text{BOOLEAN} \\
\text{ActiveStatus} \\
\text{State} \\
\text{Transition} \\
x: \text{State} \\
n: \text{Transition} \\
x.\text{activeS}=T \\
\forall y: \text{State} \quad y.\text{activeS}=T \Rightarrow y=x \\
n.\text{activeT}=T \\
\forall z: \text{Transition} \quad z.\text{activeT}=T \Rightarrow z=n
\]

\[\text{[INPORT]}\]
\[\text{[OUTPORT]}\]

The states can manipulate data within three types of actions:

\[\text{STATE\_ACTIONS::=}\text{entry}\mid\text{during}\mid\text{exit}\]
The in-ports, out-ports, states and transitions of the ACC:

<table>
<thead>
<tr>
<th>ACCModel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
</tr>
<tr>
<td><strong>Off</strong></td>
</tr>
<tr>
<td><strong>StandBy</strong></td>
</tr>
<tr>
<td><strong>CruiseByVelocity</strong></td>
</tr>
<tr>
<td><strong>CruiseByDistance</strong></td>
</tr>
<tr>
<td><strong>CruiseByVelocityToStandby</strong></td>
</tr>
<tr>
<td><strong>CruiseByDistanceToStandby</strong></td>
</tr>
<tr>
<td><strong>CruiseByDistanceToCruiseByDistance</strong></td>
</tr>
<tr>
<td><strong>CruiseByVelocityToCruiseByDistance</strong></td>
</tr>
<tr>
<td><strong>CruiseByDistanceToCruiseByVelocity</strong></td>
</tr>
<tr>
<td><strong>CruiseByDistance</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>InP</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InitSetSpeed</strong></td>
</tr>
<tr>
<td><strong>TargetPresent</strong></td>
</tr>
<tr>
<td><strong>MaxAllowedVel</strong></td>
</tr>
<tr>
<td><strong>PedalFlag</strong></td>
</tr>
<tr>
<td><strong>TargetVehicleVel</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>OutP</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ActualVel</strong></td>
</tr>
<tr>
<td><strong>ActualPedalFlag</strong></td>
</tr>
<tr>
<td><strong>DangerousBrakingAlarm</strong></td>
</tr>
</tbody>
</table>

The in-ports and out-ports read in or write out signals.

[SIGNAL]
The test harness for the ACC includes the SUT, a Test Verdicts model and the Test Generator model.

\[\text{ACCTestHarness} \]
\[\text{ACCModel} \]
\[\text{TestVerdicts} \]
\[\text{TestGenerator} \]

The Test Verdicts, compares the actual outputs of the SUT to their expected values.

\[\text{TestVerdicts} \]
\[\text{InputSignal, RefSignal: SIGNAL} \]
\[\text{TestResult: BOOLEAN} \]
\[\text{InputSignal} \neq \text{RefSignal} \Rightarrow \text{TestResult} = F \]

Aspects of the test generator relevant for my research (such as the data partition), are specified later in the document. The full Z specification of the Test Generator has been omitted for the sake of brevity.

\[\text{TestGenerator} \]

The ACC in-ports and the type of signals they have:

\[\text{ACC\_InPortsData} \]
\[\text{InS\_InitSetSpeed, InS\_Step: Z} \]
\[\text{InS\_MaxAllowedVel, InS\_MinAllowedVel: Z} \]
\[\text{InS\_HeadWay, InS\_TargetVehicleVel: Z} \]
\[\text{InS\_TargetPresent: BOOLEAN} \]
\[\text{InS\_LeverPos: LEVERPOSITION} \]
\[\text{InS\_PedalFlag: PEDALFLAG} \]

Initialisation of the in-ports data:

\[\text{InitACC\_InPortsData} \]
\[\text{ACC\_InPortsData} \]
\[\text{InS\_InitSetSpeed}=0 \]
\[\text{InS\_Step}=1 \]
\[\text{InS\_MaxAllowedVel}=69 \]
\[\text{InS\_MinAllowedVel}=0 \]
\[\text{InS\_HeadWay}=10 \]
\[\text{InS\_TargetPresent}=T \]
\[\text{InS\_LeverPos}=0 \]
\[\text{InS\_PedalFlag}=0 \]
APPENDIX A PARTIAL Z SPECIFICATION FOR AN ADAPTIVE CRUISE CONTROL

The ACC out-ports and the type of signals they have.

\[
\begin{align*}
\text{ACC\_OutPortsData} & \\
\text{OutS\_ActualVel, OutS\_TargetVelocity: } & \mathbb{Z} \\
\text{OutS\_LeverPos: } & \text{LEVERPOSITION} \\
\text{OutS\_ActualPedalFlag: } & \text{PEDALFLAG} \\
\text{OutS\_NotAllowedVelFlag: } & \text{BOOLEAN} \\
\text{OutS\_DangerousBrakingAlarm: } & \text{BOOLEAN}
\end{align*}
\]

Initialisation of the out-ports data:

\[
\begin{align*}
\text{InitACC\_OutPortsData} & \\
\text{ACC\_OutPortsData} & \\
\text{OutS\_ActualVel}=0 \\
\text{OutS\_TargetVelocity}=0 \\
\text{OutS\_LeverPos}=0 \\
\text{OutS\_ActualPedalFlag}=0 \\
\text{OutS\_NotAllowedVelFlag}=F \\
\text{OutS\_DangerousBrakingAlarm}=F
\end{align*}
\]

The SUT interface signals to the harness:

\[
\begin{align*}
\text{SystemACCInterfaceToHarness} & \\
\text{ACC\_InPortsData} \\
\text{ACC\_OutPortsData}
\end{align*}
\]
Algorithms used to manipulate data within states and/or transitions

\[ F_{\text{ncCalcVel}} \]
\[ f_{\text{CalcVel}}: \text{VELOCITY} \times \text{PEDALFLAG} \times \text{LEVERPOSITION} \rightarrow \text{VELOCITY} \times \text{BOOLEAN} \]
\[ \text{ActualVel, InitSetSpeed: VELOCITY} \]
\[ \text{AllowedVel: } \emptyset \times \text{VELOCITY} \]
\[ \text{Step, MinAllowedVel, MaxAllowedVel: VELOCITY} \]
\[ \text{NotAllowedVel: BOOLEAN} \]
\[ a: \text{PEDALFLAG} \]
\[ b: \text{LEVERPOSITION} \]
\[ \forall x: \text{VELOCITY} \bullet ((x, a, b) \in \text{dom } f_{\text{CalcVel}} \land x \in \text{AllowedVel} \\
\land a \neq 1) \Rightarrow (\text{ActualVel} = \text{InitSetSpeed} \\
\land \text{NotAllowedVel} = F \land (\text{ActualVel, NotAllowedVel}) \in \text{ran } f_{\text{CalcVel}}) \]
\[ \forall x: \text{VELOCITY} \bullet ((x, a, b) \in \text{dom } f_{\text{CalcVel}} \land x \in \text{AllowedVel} \land a \neq 1 \land \land b = 2 \land \land \text{MaxAllowedVel} \leftarrow \text{Step} \Rightarrow (\text{ActualVel} = x \land \text{NotAllowedVel} = F \land (\text{ActualVel, NotAllowedVel}) \in \text{ran } f_{\text{CalcVel}}) \]
\[ \forall x: \text{VELOCITY} \bullet ((x, a, b) \in \text{dom } f_{\text{CalcVel}} \land x \in \text{AllowedVel} \land a \neq 1 \land \land b = 3 \land \text{MinAllowedVel} \leftarrow \text{Step} \Rightarrow (\text{ActualVel} = x \land \text{NotAllowedVel} = F \land (\text{ActualVel, Not Allowed Vel}) \in \text{ran } f_{\text{CalcVel}}) \]
\[ \forall x: \text{VELOCITY} \bullet ((x, a, b) \in \text{dom } f_{\text{CalcVel}} \land x \in \text{AllowedVel} \land a \neq 1 \land \land b = 4 \Rightarrow (\text{ActualVel} = x \land \text{NotAllowedVel} = F \land (\text{ActualVel, NotAllowedVel}) \in \text{ran } f_{\text{CalcVel}}) \]

\[ F_{\text{ncCalcDist}} \]
\[ f_{\text{CalcDistance}}: \text{VELOCITY} \times \text{VELOCITY} \rightarrow \text{VELOCITY} \times \text{BOOLEAN} \]
\[ \text{ActualVel: VELOCITY} \]
\[ \text{AllowedVel: } \emptyset \times \text{VELOCITY} \]
\[ \text{NotAllowedVel, Alarm: BOOLEAN} \]
\[ \forall x, y: \text{VELOCITY} \bullet ((x, y) \in \text{dom } f_{\text{CalcDistance}} \land x \in \text{AllowedVel} \land x \leq y) \Rightarrow (\text{ActualVel} = x \land \text{Alarm} = F \land (\text{ActualVel, Alarm}) \in \text{ran } f_{\text{CalcDistance}}) \]
\[ \forall x, y: \text{VELOCITY} \bullet ((x, y) \in \text{dom } f_{\text{CalcDistance}} \land x \in \text{AllowedVel} \land x \leq y \land (x - y) \div ((x \times x) \div 10) \div y < 11) \Rightarrow (\text{ActualVel, Alarm}) \in \text{ran } f_{\text{CalcDistance}} \]
\[ \forall x, y: \text{VELOCITY} \bullet ((x, y) \in \text{dom } f_{\text{CalcDistance}} \land x \in \text{AllowedVel} \land x \leq y \land (x - y) \div (((x \times x) \div 10) \div x) \times 11) \Rightarrow (\text{ActualVel} = (x + (x - y) \div (((x \times x) \div 10) \div x) \times (10 \times x)) \land \text{Alarm} = F \land (\text{ActualVel, Alarm}) \in \text{ran } f_{\text{CalcDistance}}) \]
APPENDIX A PARTIAL Z SPECIFICATION FOR AN ADAPTIVE CRUISE CONTROL

The states of the ACC

State Off:

\[ \text{StateOff} \]

StateOff: \text{STATE}

StateOffActive: \text{BOOLEAN}

action: \text{STATE_ACTIONS}

ActualVel, InitSetSpeed, Out_TargetVelocity, TargetVelocity: \text{VELOCITY}

LeverPos, Out_LeverPos: \text{LEVERPOSITION}

PedalFlag, Out_PedalFlag, Out_ActualPedalFlag: \text{PEDALFLAG}

LeverPos=0

(action=entry⇒StateOffActive=T∧ActualVel=InitSetSpeed∧Out_LeverPos=LeverPos∧
Out_ActualPedalFlag=PedalFlag∧Out_TargetVelocity=TargetVelocity)∨

(action=exit⇒StateOffActive=F)

State Standby:

\[ \text{StateStandby} \]

StateStandby: \text{STATE}

StateStandbyActive: \text{BOOLEAN}

action: \text{STATE_ACTIONS}

ActualVel, InitSetSpeed: \text{VELOCITY}

LastVelocity, Out_TargetVelocity, TargetVelocity: \text{VELOCITY}

NotAllowedVel: \text{BOOLEAN}

LeverPos, Out_LeverPos: \text{LEVERPOSITION}

PedalFlag, Out_ActualPedalFlag: \text{PEDALFLAG}

LeverPos=0

(action=entry⇒StateStandbyActive=T∧ActualVel=InitSetSpeed)\∨

(action=exit⇒StateStandbyActive=F)\∧

(LastVelocity=ActualVel∧NotAllowedVel=F∧Out_LeverPos=LeverPos∧
Out_ActualPedalFlag=PedalFlag∧Out_TargetVelocity=TargetVelocity)
APPENDIX A PARTIAL Z SPECIFICATION FOR AN ADAPTIVE CRUISE CONTROL

State CruiseByVelocity:

\[ \text{CruiseByVel} \]
\[ \text{CruiseByVelActive: BOOLEAN} \]
\[ \text{FncCalcVel} \]
\[ \text{action: STATE ACTIONS} \]
\[ \text{LastVelocity, ActualVel, ActualVel': VELOCITY} \]
\[ \text{TargetVelocity, Out_TargetVelocity: VELOCITY} \]
\[ \text{NotAllowedVelFlag, NotAllowedVelFlag': BOOLEAN} \]
\[ \text{PedalFlag, Out_ActualPedalFlag: PEDALFLAG} \]
\[ \text{LeverPos, Out_LeverPos: LEVERPOSITION} \]
\[ (\text{action}=\text{entry} \Rightarrow \text{StateCruiseByVelActive}=T) \land \]
\[ (\text{action}=\text{exit} \Rightarrow \text{LastVelocity}=\text{ActualVel} \land \text{StateCruiseByVelActive}=F) \]
\[ (\text{ActualVel}', \text{NotAllowedVelFlag}')=\text{CalcVel}(\text{ActualVel}, \text{PedalFlag}, \text{LeverPos}) \]
\[ \text{Out_LeverPos}=\text{LeverPos} \land \text{Out_ActualPedalFlag}=\text{PedalFlag} \]
\[ \text{Out_TargetVelocity}=\text{TargetVelocity} \]

State CruiseByDistance:

\[ \text{CruiseByDist} \]
\[ \text{CruiseByDistActive: BOOLEAN} \]
\[ \text{action: STATE ACTIONS} \]
\[ \text{ActualVel, LastVelocity: VELOCITY} \]
\[ \text{Out_TargetVelocity, TargetVelocity: VELOCITY} \]
\[ \text{ValidVel: P VELOCITY} \]
\[ \text{LeverPos, Out_LeverPos: LEVERPOSITION} \]
\[ \text{PedalFlag, Out_ActualPedalFlag: PEDALFLAG} \]
\[ (\text{action}=\text{entry} \Rightarrow \text{StateCruiseByDistActive}=T \land \text{ActualVel}\in\text{ValidVel}) \land \]
\[ (\text{action}=\text{exit} \Rightarrow \text{LastVelocity}=\text{ActualVel} \land \text{Out_LeverPos}=\text{LeverPos} \land \]
\[ (\text{Out_ActualPedalFlag}=\text{PedalFlag} \land \text{Out_TargetVelocity}=\text{TargetVelocity} \land \]
\[ \text{StateCruiseByDistActive}=F) \]

State-transitions of the ACC

Default transition (the Statechart’s entry point):

\[ \text{TrIntoOff} \]
\[ \Delta \text{StateOff} \]
\[ \text{ActiveVel, InitSetSpeed: VELOCITY} \]
\[ \text{ActiveVel}=\text{InitSetSpeed} \]
\[ \text{StateOffActive}'=T \]
Transition from state *Off* to state *Standby*:

\[ \text{TrStateOffToStandby} \]
\[
\Delta \text{StateOff} \\
\Delta \text{StateStandby} \\
\text{ActualVel: VELOCITY} \\
\text{ValidVel: } \not\exists \text{VELOCITY} \\
\text{LeverPos: LEVERPOSITION} \\
\]
\[
\text{ActualVel} \in \text{ValidVel} \\
\text{LeverPos} \neq 0 \\
\text{StateStandbyActive}'=T \\
\text{ActualVel}'=\text{ActualVel} \\
\]

Transition from state *Standby* to state *Off*:

\[ \text{TrStateStandbyToOff} \]
\[
\Delta \text{StateStandby} \\
\Delta \text{StateOff} \\
\text{ActualVel, LastVelocity: VELOCITY} \\
\text{LeverPos: LEVERPOSITION} \\
\]
\[
\text{StateOffActive}'=T \\
\text{LeverPos} = 0 \\
\text{ActualVel} = \text{LastVelocity} \\
\]

Transition from state *Standby* back into *Standby*:

\[ \text{TrStateStandbyToSelf} \]
\[
\Delta \text{StateStandby} \\
\text{ActualVel: VELOCITY} \\
\text{ValidVel: } \not\exists \text{VELOCITY} \\
\text{LeverPos: LEVERPOSITION} \\
\text{PedalFlag: PEDALFLAG} \\
\]
\[
\text{StateStandbyActive}'=T \\
\text{ActualVel} \in \text{ValidVel} \\
\text{LeverPos} \neq 0 \land \text{PedalFlag} = 2 \land \text{LeverPos} \neq 0 \\
\text{ActualVel} = \text{LastVelocity} \\
\]
Transition from state \textit{Standby} to state \textit{CruiseByVelocity}:

\[
\text{TrStateStandbyToStateCruiseByVel}
\]
\[
\Delta \text{StateStandby}
\]
\[
\Delta \text{StateCruiseByVel}
\]
\[
\text{FncCalcVel}
\]
\[
\text{ActualVel: VELOCITY}
\]
\[
\text{ValidVel: } \not\in \text{ VELOCITY}
\]
\[
\text{LeverPos: LEVERPOSITION}
\]
\[
\text{PedalFlag: PEDALFLAG}
\]
\[
\text{TargetPresent, NotAllowedVelFlag: BOOLEAN}
\]
\[
\text{StateCruiseByVelActive}' = T
\]
\[
\text{ActualVel} \in \text{ValidVel}
\]
\[
\text{LeverPos} \neq 0 \lor \text{LeverPos} \neq 1 \land \text{TargetPresent} = F \land \text{PedalFlag} \neq 2
\]
\[
(\text{ActualVel}', \text{NotAllowedVelFlag}') = \text{fCalcVel}(\text{ActualVel}, \text{PedalFlag}, \text{LeverPos})
\]

Transition from state \textit{CruiseByVelocity} to state \textit{Standby}:

\[
\text{TrStateCruiseByVelToStateStandby}
\]
\[
\Delta \text{StateCruiseByVel}
\]
\[
\Delta \text{StateStandby}
\]
\[
\text{ActualVel: VELOCITY}
\]
\[
\text{ValidVel: } \not\in \text{ VELOCITY}
\]
\[
\text{PedalFlag: PEDALFLAG}
\]
\[
\text{LeverPos: LEVERPOSITION}
\]
\[
\text{NotAllowedVel, TargetPresent: BOOLEAN}
\]
\[
\text{StateStandbyActive}' = T
\]
\[
\text{ActualVel} \in \text{ValidVel}
\]
\[
\text{PedalFlag} = 2 \lor \text{LeverPos} = 0 \lor \text{LeverPos} = 1
\]
\[
\text{NotAllowedVel} = T \land \text{TargetPresent} = F
\]

Transition from state \textit{CruiseByVelocity} back into state \textit{CruiseByVelocity}:

\[
\text{TrStateCruiseByVelToSelf}
\]
\[
\Delta \text{StateCruiseByVel}
\]
\[
\text{FncCalcVel}
\]
\[
\text{ActualVel: VELOCITY}
\]
\[
\text{ValidVel: } \not\in \text{ VELOCITY}
\]
\[
\text{LeverPos: LEVERPOSITION}
\]
\[
\text{PedalFlag: PEDALFLAG}
\]
\[
\text{NotAllowedVel, TargetPresent: BOOLEAN}
\]
\[
\text{StateCruiseByVelActive}' = T
\]
\[
\text{ActualVel} \in \text{ValidVel}
\]
\[
\text{LeverPos} = 2 \lor \text{LeverPos} = 3 \lor \text{LeverPos} = 4 \land \text{TargetPresent} = F
\]
\[
\text{PedalFlag} = 2 \land \text{NotAllowedVel} = F
\]
\[
(\text{ActualVel}', \text{NotAllowedVelFlag}') = \text{fCalcVel}(\text{ActualVel}, \text{PedalFlag}, \text{LeverPos})
\]
Transition from state \textit{CruiseByVelocity} to state \textit{CruiseByDistance}:\newline
\begin{itemize}
  \item $\Delta \text{StateCruiseByDist}$
  \item $\Delta \text{StateCruiseByVel}$
  \item $FncCalcVel$
  \item $ActualVel, TargetVel: VELOCITY$
  \item $\text{ValidVel}: \mathbb{P} \text{VELOCITY}$
  \item $PedalFlag: \text{PEDALFLAG}$
  \item $LeverPos: \text{LEVERPOSITION}$
  \item $\text{TargetPresent, Alarm}: \text{BOOLEAN}$
\end{itemize}
\begin{itemize}
  \item $\text{StateCruiseByDistActive}' = T$
  \item $ActualVel \in \text{ValidVel}$
  \item $TargetPresent = T \land PedalFlag \neq 2 \land LeverPos \neq 0 \land LeverPos \neq 1$
  \item $(ActualVel', \text{NotAllowedVelFlag}') = fCalcVel(ActualVel, PedalFlag, LeverPos)$
\end{itemize}

Transition from state \textit{CruiseByDistance} to state \textit{CruiseByVelocity}:\newline
\begin{itemize}
  \item $\Delta \text{StateCruiseByVel}$
  \item $\Delta \text{StateCruiseByDist}$
  \item $FncCalcVel$
  \item $ActualVel: VELOCITY$
  \item $\text{ValidVel}: \mathbb{P} \text{VELOCITY}$
  \item $LeverPos: \text{LEVERPOSITION}$
  \item $\text{TargetPresent, Alarm, NotAllowedVelFlag}: \text{BOOLEAN}$
\end{itemize}
\begin{itemize}
  \item $\text{StateCruiseByVelActive}' = T$
  \item $ActualVel \in \text{ValidVel}$
  \item $LeverPos \neq 0 \land LeverPos \neq 1 \land TargetPresent = F$
  \item $PedalFlag = 2 \land Alarm = F$
  \item $(ActualVel', \text{NotAllowedVelFlag}') = fCalcVel(ActualVel, PedalFlag, LeverPos)$
\end{itemize}

Transition from state \textit{CruiseByDistance} back into state \textit{CruiseByDistance}:
APPENDIX A PARTIAL Z SPECIFICATION FOR AN ADAPTIVE CRUISE CONTROL

<table>
<thead>
<tr>
<th>Transition from state CruiseByDistance to state Standby:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Tr_{\text{StateCruiseByDistToStateStandby}} )</td>
</tr>
<tr>
<td>( \Delta \text{StateCruiseByDist} )</td>
</tr>
<tr>
<td>( \Delta \text{StateStandby} )</td>
</tr>
<tr>
<td>( \text{ActualVel}: \text{VELOCITY} )</td>
</tr>
<tr>
<td>( \text{LeverPos}: \text{LEVERPOSITION} )</td>
</tr>
<tr>
<td>( \text{Alarm}: \text{BOOLEAN} )</td>
</tr>
<tr>
<td>( \text{StateStandbyActive}'=\text{T} )</td>
</tr>
<tr>
<td>( \text{LeverPos}=0 \lor \text{LeverPos}=1 \lor \text{PedalFlag}=2 \lor \text{Alarm}=\text{T} )</td>
</tr>
<tr>
<td>( \text{ActualVel}'=\text{ActualVel} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition from state Standby to state CruiseByDistance:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Tr_{\text{StateStandbyToStateCruiseByDist}} )</td>
</tr>
<tr>
<td>( \Delta \text{StateStandby} )</td>
</tr>
<tr>
<td>( \Delta \text{StateCruiseByDist} )</td>
</tr>
<tr>
<td>( \text{FncCalcDist} )</td>
</tr>
<tr>
<td>( \text{ActualVel}, \text{TargetVelocity}: \text{VELOCITY} )</td>
</tr>
<tr>
<td>( \text{ValidVel}: \text{P} \text{ VELOCITY} )</td>
</tr>
<tr>
<td>( \text{LeverPos}: \text{LEVERPOSITION} )</td>
</tr>
<tr>
<td>( \text{PedalFlag}: \text{PEDALFLAG} )</td>
</tr>
<tr>
<td>( \text{TargetPresent}, \text{Alarm}: \text{BOOLEAN} )</td>
</tr>
<tr>
<td>( \text{StateCruiseByDistActive}'=\text{T} )</td>
</tr>
<tr>
<td>( \text{ActualVel} \in \text{ValidVel} )</td>
</tr>
<tr>
<td>( \text{LeverPos} \neq 0 \land \text{LeverPos} \neq 1 \land \text{TargetPresent}=\text{T} \land \text{PedalFlag}=2 )</td>
</tr>
<tr>
<td>( (\text{ActualVel}', \text{Alarm})=\text{fCalcDistance}(\text{ActualVel}, \text{TargetVelocity}) )</td>
</tr>
</tbody>
</table>
APPENDIX B – Illustration of CTM\textsubscript{CONTROL} within the Formalization of a Reactive Testing Process

Illustration of using CTM\textsubscript{CONTROL} within the Formalization of a Reactive Testing Process

The Formalization of a Reactive Testing Process proposed in Chapter 4 of this thesis, assumes the existence of a formal specification for the SUT, hence a (partial) formal specification for the SUT, written in Z notation, is presented in this appendix.

The workflow of the FRTP starts with capturing formally the test objectives, it follows with the formalization of the test reactivity, the formalization of the partition heuristics, and the formalization of test data generation and selection\textsuperscript{51}.

1. Formalization of Test Objectives

The test objective for this case study is to measure the MC/DC coverage delivered by CTM\textsubscript{CONTROL} in comparison to that delivered by the classical CTM. This objective can be captured formally in Z notation. As specified in the Z schema below, the test harness can be executed in two modes: CTM or CTM\_CONTROL. Here coverage\textsubscript{A}, coverage\textsubscript{B} and coverage\textsubscript{!} are coverage reports which provide the percentage of MC/DC coverage achieved by the testing process. The Boolean rep\textsubscript{!} is set to true if the CTM\_CONTROL mode achieves a higher MC/DC coverage than the CTM mode. Otherwise rep\textsubscript{!} is set to false.

\footnotesize

\begin{align*}
\text{TESTMODE}::=& \text{CTM}\text{|CTM\_CONTROL}
\end{align*}

\textsuperscript{51} Selection in this instance is understood as the choice of which test sequences are ran (i.e. test control).
MeasureMCDC_Coverage

\text{ACCTestHarness}
\text{testMode: TESTMODE}
\text{coverage!: Z}
\text{coverageA, coverageB: Z}
\text{rep!: BOOLEAN}

\begin{align*}
testMode=\text{CTM} & \Rightarrow \text{coverage!} \neq \text{coverageA} \\
testMode=\text{CTM\_CONTROL} & \Rightarrow \text{coverage!} \neq \text{coverageB} \\
\text{coverageB} > \text{coverageA} & \Rightarrow \text{rep!} = T \\
\text{coverageB} \leq \text{coverageA} & \Rightarrow \text{rep!} = F
\end{align*}

2. **Formalization of Test Reactivity**

In the proposed test reactivity taxonomy, as depicted in Figure 53, test reactivity dimensions fall into three major categories:

4. Coverage criteria related.
5. Input, output and temporal space related.

The test reactivity, which was considered in this case study falls into the Signal Properties of the Input, output and temporal space related dimension of the proposed test reactivity taxonomy (Fig. 68).
For the case study, two reactive scenarios were captured and implemented in the M/S/S environment.

**Reactive Scenario No. 1:** The vehicle is travelling at a speed ($\text{ActualVel?}$) of over 40 [m/s] (144 [km/h]). The ACC is turned on and the radar picks up another vehicle that abruptly cuts in front of the first vehicle ($\text{TargetPresent?}=T$). The other vehicle is travelling at a speed ($\text{TargetVel?}$) lower than 17 [m/s] (61.2 [km/h]).

The next test after this scenario is observed, should investigate what happens if the vehicle in front proceeds to brake abruptly and come to a full stop. In other words, the test harness should execute the test case designed to be triggered if this scenario (reactive conceptual state) is detected. The test case designed for this test scenario specifies that the first vehicle’s
velocity should be 40 m/s \((\text{NextActualVel}!=40)\); the velocity of the vehicle in front should be 0 m/s \((\text{NextTargetVel}!=0)\); the ACC lever position should be in position 2 \((\text{LeverPos}=2)\); and, the driver should press the brake’s pedal \((\text{PedalFlag}=2)\).

<table>
<thead>
<tr>
<th>ReactTestAbstState1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActualVel?, TargetVel?, NextActualVel!, NextTargetVel!: VELOCITY</td>
</tr>
<tr>
<td>TargetPresent?: BOOLEAN</td>
</tr>
<tr>
<td>LeverPos: LEVERPOSITION</td>
</tr>
<tr>
<td>PedalFlag: PEDALFLAG</td>
</tr>
</tbody>
</table>

\[
\text{ActualVel}? > 40 \land \text{TargetPresent}? = T \land \text{TargetVel}? < 17 \Rightarrow (\text{NextActualVel}!=40 \land \text{NextTargetVel}!=0 \land \text{LeverPos}=2 \land \text{PedalFlag}=2)
\]

**Reactive Scenario No. 2:** The vehicle is travelling at a speed \(V_A\) \((\text{ActualVel}?\) of over 40 [m/s]. The ACC is turned on and the radar picks up another vehicle \((\text{TargetPresent}?=T)\) that is travelling at a speed \(V_B\), which is around the same speed as that of our vehicle \(((\text{ActualVel}?−1)\land \text{TargetVel}?<(\text{ActualVel}?+1))\). The next test after this scenario is observed, should investigate what happens if our vehicle has matched the previous velocity of the vehicle in front \((\text{NextActualVel}!=\text{TargetVel}?\)\), while the vehicle in front has accelerated and has increased its velocity by 10 m/s \((\text{TargetVel}?+10)\). Additionally the ACC lever should be in position 2 \((\text{LeverPos}=2)\) and the driver should not press any pedals \((\text{PedalFlag}=0)\).

<table>
<thead>
<tr>
<th>ReactTestAbstState2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActualVel?, TargetVel?, NextActualVel!, NextTargetVel!: VELOCITY</td>
</tr>
<tr>
<td>TargetPresent?: BOOLEAN</td>
</tr>
<tr>
<td>LeverPos: LEVERPOSITION</td>
</tr>
<tr>
<td>PedalFlag: PEDALFLAG</td>
</tr>
</tbody>
</table>

\[
\text{ActualVel}? > 40 \land \text{TargetPresent}? = T \land \text{TargetVel}?<(\text{ActualVel}?−1)\land \text{TargetVel}?<(\text{ActualVel}?+1) \Rightarrow (\text{NextActualVel}!=\text{TargetVel}!\land \text{NextTargetVel}=(\text{TargetVel}!+10)\land \text{LeverPos}=2 \land \text{PedalFlag}=0)
\]
3. Formalization of the Partition Heuristics, and Test Data Generation

The testing activities discussed here depict a point during the test process where the SUT parameters, which are considered as of interest are: \textit{InitSetSpeed, TargetVehicleVel, TargetPresent, LeverPos} and \textit{PedalFlag}.

The data partition heuristics are formalized and test data values are selected for each of these five parameters. The formalization is presented both in Z notation and via a tabular format for illustration purposes. Some of the data partitions have been represented with more than one data value in order to illustrate the formalization of weighted partitions.

The formalization is detailed for each parameter separately.

**Formalization of the data partitions and the parameterization of the partitions for InitSetSpeed.**

\textit{InitSetSpeed}: The velocity of the current vehicle where the ACC under test is installed.

Here the partitions are \textit{Min\_Vel} for velocities \(i\in[0-8) \text{ km/h};\) \textit{Low\_Vel} for velocities \(j\in[8-14) \text{ km/h};\) \textit{Med\_Vel} for velocities \(k\in[14-18);\) \textit{High\_Vel} for velocities \(l\in[18-43);\) and \textit{Max\_Vel} for velocities \(m\in[43-70).\)

**Formalization in Z notation:**

\[
\begin{align*}
\text{InitSetVelFormalPartitions} & \\
\text{Min\_Vel, Low\_Vel, Med\_Vel: } & \mathbb{P} \text{ VELOCITY} \\
\text{High\_Vel, Max\_Vel: } & \mathbb{P} \text{ VELOCITY} \\
\forall i: \text{ VELOCITY}\bullet i \in \text{Min\_Vel} \Rightarrow i \geq 0 \land i < 8 \\
\forall j: \text{ VELOCITY}\bullet j \in \text{Low\_Vel} \Rightarrow j \geq 8 \land j < 14 \\
\forall k: \text{ VELOCITY}\bullet k \in \text{Med\_Vel} \Rightarrow k \geq 14 \land k < 18 \\
\forall l: \text{ VELOCITY}\bullet l \in \text{High\_Vel} \Rightarrow l \geq 18 \land l < 43 \\
\forall m: \text{ VELOCITY}\bullet m \in \text{Max\_Vel} \Rightarrow m \geq 43 \land m < 70
\end{align*}
\]
**Formalization in a tabular format:**

**Table 33: InitSetSpeed (Vehicle Velocity)**

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>Min_Vel</th>
<th>Low_Vel</th>
<th>Med_Vel</th>
<th>High_Vel</th>
<th>Max_Vel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min_Vel</td>
<td>[0-8]</td>
<td>(8.6 – 13.9)</td>
<td>(13.9-18)</td>
<td>[18-43)</td>
<td>[43-69.4)</td>
</tr>
<tr>
<td>Low_Vel</td>
<td>W=1;</td>
<td>W=3;</td>
<td>W=1;</td>
<td>W=3;</td>
<td>W=1;</td>
</tr>
<tr>
<td>Sf=interpolated</td>
<td>Sf=interpolated</td>
<td>Sf=interpolated</td>
<td>Sf: interpolated</td>
<td>Sf: interpolated</td>
<td></td>
</tr>
<tr>
<td>Med_Vel</td>
<td>(0)</td>
<td>(8.6,8.9,9.2)</td>
<td>(16.7)</td>
<td>(41.9,42,42.2)</td>
<td>(69.4)</td>
</tr>
<tr>
<td>High_Vel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max_Vel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TargetVehicleVelFormalPartitions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Formalization of the data partitions and the parameterization of the partitions for TargetVehicleVel.**

**TargetVehicleVel:** The velocity of the vehicle ahead, which has been picked up by the radar.

Here the partitions are Min_Vel for velocities $i \in [0-8)$ km/h; Low_Vel for velocities $j \in [8-14)$ km/h; Med_Vel for velocities $k \in [14-18)$; High_Vel for velocities $l \in [18-43)$; and Max_Vel for velocities $m \in [43-70)$.

**Formalization in Z notation:**

```
TargetVehicleVelFormalPartitions
Min_Vel, Low_Vel, Med_Vel, High_Vel, Max_Vel: ⊆ VELOCITY
∀i: VELOCITY • i ∈ Min_Vel ⇒ i ≥ 0 ∧ i < 8
∀j: VELOCITY • j ∈ Low_Vel ⇒ j ≥ 8 ∧ j < 14
∀k: VELOCITY • k ∈ Med_Vel ⇒ k ≥ 14 ∧ k < 18
∀l: VELOCITY • l ∈ High_Vel ⇒ l ≥ 18 ∧ l < 43
∀m: VELOCITY • m ∈ Max_Vel ⇒ m ≥ 43 ∧ m < 70
```
Formalization in a tabular format:

Table 34: TargetVehicleVel (Velocity of the vehicle ahead)

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>Min_Vel</th>
<th>Low_Vel</th>
<th>Med_Vel</th>
<th>High_Vel</th>
<th>Max_Vel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [m/s]</td>
<td>[0]</td>
<td>[8.6 – 13.9)</td>
<td>[13.9-18)</td>
<td>[18-43)</td>
<td>[43-69.4)</td>
</tr>
<tr>
<td>Weighting factor and signal function</td>
<td>W=1;</td>
<td>W=3;</td>
<td>W=1;</td>
<td>W=3;</td>
<td>W=1;</td>
</tr>
<tr>
<td></td>
<td>Sf:interpolated</td>
<td>Sf:interpolated</td>
<td>Sf:interpolated</td>
<td>Sf:interpolated</td>
<td>Sf:interpolated</td>
</tr>
<tr>
<td>Data selection [m/s]</td>
<td>(0)</td>
<td>(8.6, 8.9, 9.2)</td>
<td>(16.7)</td>
<td>(41.9, 42, 42.2)</td>
<td>(69.4)</td>
</tr>
</tbody>
</table>

Formalization of the data partitions and the parameterization of the partitions for TargetPresent.

**TargetPresent**: A Boolean parameter, which is true if the radar has identified a potential obstacle in the current vehicle’s path. The equivalence partitions are Vehicle_Ahead and No_Vehicle_Ahead.

**Formalization in Z notation**:

```
TargetPresentFormalPartitions
Vehicle_Ahead, No_Vehicle_Ahead: ℙ BOOLEAN
∀i: BOOLEAN•i∈Vehicle_Ahead⇒i=T
∀j: BOOLEAN•j∈No_Vehicle_Ahead⇒j=F
```
Formalization in a tabular format:

Table 35: TargetPresent (Vehicle Ahead detected flag)

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>Vehicle_Ahead</th>
<th>No_Vehicle_Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [m/s]</td>
<td>[1]</td>
<td>[0]</td>
</tr>
<tr>
<td>Weighting factor and signal function</td>
<td>W=1; Sf= step function</td>
<td>W=1; Sf= step function</td>
</tr>
<tr>
<td>Data selection [m/s]</td>
<td>(1)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

Formalization of the data partitions and the parameterization of the partitions for TargetPresent.

LeverPos: The ACC lever has 5 positions: 0=Off, 1=Neutral, 2=Set+, 3=Set-, 4=Resume). The equivalence partitions are Off [0], Neutral [1], SetPlus [2], SetMinus [3] and Resume [4].

Formalization in Z notation:

\[
\text{LeverPosFormalPartitions:} \quad \forall i: \mathbb{N} \ni \text{Off} \Rightarrow i = 0 \\
\forall j: \mathbb{N} \ni \text{Neutral} \Rightarrow j = 1 \\
\forall k: \mathbb{N} \ni \text{SetPlus} \Rightarrow k = 2 \\
\forall l: \mathbb{N} \ni \text{SetMinus} \Rightarrow l = 3 \\
\forall m: \mathbb{N} \ni \text{Resume} \Rightarrow m = 4
\]
**Formalization in a tabular format:**

Table 36: LeverPos (ACC lever position)

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>Off</th>
<th>Neutral</th>
<th>Set+</th>
<th>Set-</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [m/s]</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
<tr>
<td>Weighting factor and signal function W=1; Sf=step function</td>
<td>W=3; Sf= step function</td>
<td>W=1; Sf= step function</td>
<td>W=3; Sf: step function</td>
<td>W=1; Sf: step function</td>
<td></td>
</tr>
<tr>
<td>Data selection [m/s]</td>
<td>(0)</td>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
</tr>
</tbody>
</table>
Formalization in a tabular format:

Table 37: Pedal flag (Pedal pressed flag)

<table>
<thead>
<tr>
<th>Abstract equivalence partitions</th>
<th>No_Pedal_Pressed</th>
<th>Acceleration_Pedal_Pressed</th>
<th>Brake_Pedal_Pressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized Partitions sets [m/s]</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
<tr>
<td>Weighting factor and signal function</td>
<td>W=1; Sf=step function</td>
<td>W=3; Sf= step function</td>
<td>W=1; Sf= step function</td>
</tr>
<tr>
<td>Data selection [m/s]</td>
<td>(0)</td>
<td>(1)</td>
<td>(2)</td>
</tr>
</tbody>
</table>

The MC/DC coverage for this case study is focused on the parameters LeverPos, PedalFlag and TargetPresent and their use in the state-transitions of the SUT. To generate the tests for the MC/DC aspect of $\text{CTM}_{\text{CONTROL}}$, all the atomic logic expressions part of the state-transition guards, which contain the above parameters, need to be considered separately. Each decision and condition is toggled between true and false. For example, the atomic logic expression LeverPos!=0, can be toggled between true and false respectively for the values of LeverPos!=(0, ¬0). Similarly, the atomic expression PedalFlag==2 can be toggled between true to false by the values of PedalFlag=(2, ¬2). The non-fully parameterized set of parameter values, which toggle all the atomic logic expressions of the transitions between true and false is depicted in Table 38:
Table 38: Non-fully parameterized set of values

<table>
<thead>
<tr>
<th>LeverPos</th>
<th>PedalFlag</th>
<th>TargetPresent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>not 0</td>
<td>not 2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>not1</td>
<td></td>
</tr>
<tr>
<td>not 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One from the possible refinements of the above representation of data into a fully parameterized set of values, is given in Table 39.

Table 39: The parameter values considered for the MC/DC

<table>
<thead>
<tr>
<th>LeverPos</th>
<th>PedalFlag</th>
<th>TargetPresent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summarizing, the test data generated from the process of parametrizing the CTM partitions and targeted MC/DC coverage so far are depicted in Table 40.
Table 40: Test data used for both test approaches

<table>
<thead>
<tr>
<th>Test data generated via a CTM approach</th>
<th>Vehicle velocity</th>
<th>(0, (8.6, 8.9, 9.2), (16.7), (41.9, 42, 42.2), (69.4))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle in front velocity</td>
<td>(0, (8.6, 8.9, 9.2), (16.7), (41.9, 42, 42.2), (69.4))</td>
<td></td>
</tr>
<tr>
<td>Vehicle ahead detected flag</td>
<td>(0, 1)</td>
<td></td>
</tr>
<tr>
<td>ACC lever position</td>
<td>(0, 1, 2, 3, 4)</td>
<td></td>
</tr>
<tr>
<td>Pedal pressed flag</td>
<td>(0, 1, 2)</td>
<td></td>
</tr>
<tr>
<td>Test data generated via the MC/DC aspect of CTM&lt;sub&gt;CONTROL&lt;/sub&gt;</td>
<td>LeverPos (ACC lever position)</td>
<td>(0, 1, 2, 3, 4)</td>
</tr>
<tr>
<td></td>
<td>PedalFlag (Pedal pressed flag)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td></td>
<td>TargetPresent (Vehicle ahead detected flag)</td>
<td>(0, 1)</td>
</tr>
</tbody>
</table>
APPENDIX C – Screenshots from the Implementation

This appendix depicts a selection of screen shots from the prototypical implementation of CTM control for the ACC case study.

A number of detailed views of the modelled systems have been omitted for clarity.
Overview of the Test Harness:
A partial view inside of the TestGenerator:
A closer look at the *TestSequenceSelector* interfaces:
Inside the Test Sequence Selector:

Logic followed:
if none of the flags capturing the reactive behavior have been set to true continue with the consecutive order of test sequences.
else jump into the desired test sequence number.
Inside Reactive Test Scenario 1:

Inside Reactive Test Scenario 2:

Difference in decimal places, i.e., RelativeDifference = 20000 translates into a 2% difference with the values with which it is being compared.
Inside TestControlModeLayer:
A view on how the tests generated automatically via MATLAB commands are loaded up and prepared for execution from the SubsystemforMCDAспект, SubsystemforfullCTM\textsubscript{CONTROL}, SubsystemforClassicCTM, and a number of other subsystems, which can execute the tests for other be-spoke test modes:
APPENDIX C SCREENSHOTS FROM THE IMPLEMENTATION

A view at the ACC captured via a Stateflow Statechart:
The graphical function \texttt{funcCalcVel()}:
The graphical function `funcCalcDistance()`:
APPENDIX D – MATLAB Code

This appendix depicts a selection of snippets of code, which was written in MATLAB’s own language to implement a number of the algorithms used in this research.
% matrix manipulation, data extraction
% G is a previously generated matrix

G = G(1, :);
columns = s(1, 2);
for row = 1:rows
    b = G(row, :);
    l = repmat(b, 3, 1);
    M = [M; l];
end;

G = M;
s = size(G);
rows = s(1, 1);
x = colon(1, rows);
t = reshape(x, [rows, 1]);

sigVel = G(:, 1);    simin1 = [t sigVel];

sigTargVel = G(:, 2);    simin2 = [t sigTargVel];

SigVehicleAhead = G(:, 3);    simin3 = [t SigVehicleAhead];

SigLeverPos = G(:, 4);    simin4 = [t SigLeverPos];

SigPedalFlag = G(:, 5);    simin5 = [t SigPedalFlag];

s1 = size(GM);
rows1 = s1(1, 1);
for row1 = 1:rows1
    b1 = GM(row1, :);
    l1 = repmat(b1, 3, 1);
    M1 = [M1; l1];
end;
GM = M1;
s2 = size(GM);
rows1 = s2(1, 1);
x1 = colon(1, rows1);
t1 = reshape(x1, [rows1, 1]);
% Generating permutations

s = size(GM);
rows = s(1,1);
M_i = GM(1,:);

for row_i = 1:rows
    temp = GM(row_i,:);
    temp1 = repmat(temp, 3, 1);
    M_i = [M_i; temp1];
end;

M_j = M_i(1,:);
rows_p = size(M_i);

for j = 1:rows_p
    temp_p = M_i(j,:);
    temp1_p = perms(temp_p);
    M_j = [M_j; temp1_p];
end;

% shape the test signal for the test harness

GM = M_j;

S1 = size(GM);
rows1 = S1(1,1);
x4 = colon(1,rows1);
t4 = reshape(x4, [rows1, 1]);

SigVehicleAheadM = GM(:, 3);
simin33 = [t4 SigVehicleAheadM];

SigLeverPosM = GM(:, 1);
simin44 = [t4 SigLeverPosM];

SigPedalFlagM = GM(:, 2);
simin55 = [t4 SigPedalFlagM];