Model-based Testing of Real-Time Embedded Systems in the Automotive Domain

vorgelegt von

Dipl. Inform., Dipl. Ing.
Justyna Zander-Nowicka

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades der

Doktorin der Ingenieurwissenschaften

– Dr.-Ing. –

Genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. –Ing. Clemens Gühmann
Berichter: Prof. Dr. –Ing. Ina Schieferdecker
Berichter: Prof. Dr. rer. nat. Ingolf Heiko Krüger

Tag der wissenschaftlichen Aussprache: 19.12.2008

Berlin, 2009
D 83

Model-based Testing of Real-Time Embedded Systems in the Automotive Domain

by

M. Sc.
Justyna Zander-Nowicka

Faculty IV – Electrical Engineering and Computer Science
Technical University Berlin
A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Engineering Science

– Eng. Sc. D. –

Accredited Dissertation

Examination Board:
Chairman: Prof. Eng. Sc. D. Clemens Gühmann

Supervisor: Prof. Eng. Sc. D. Ina Schieferdecker
Technical University Berlin, Faculty of Electrical Engineering and Computer Science

Supervisor: Prof. Dr. Ingolf Heiko Krüger
University of California, San Diego, Department of Computer Science and Engineering

Day of the Defense: December 19th, 2008

Berlin, 2009
D 83

To my parents Ewa and Georg Zander.

Technical University Berlin
Faculty IV – Electrical Engineering and Computer Science
Department for Design and Testing of Telecommunications Systems
Franklinstraße 28-29
10587 Berlin, Germany
http://www.iv.tu-berlin.de/

University of California, San Diego
Department of Computer Science and Engineering
UCSD CSE Building
9500 Gilman Drive, Dept. 0404
La Jolla, CA 92093-0404, U.S.A.
https://sosa.ucsd.edu/

Abstract

Software aspects of embedded systems are expected to have the greatest impact on industry, market and everyday life in the near future. This motivates the investigation of this field. Furthermore, the creation of consistent, reusable, and well-documented models becomes an important stage in the development of embedded systems. Design decisions that used to be made at the code level are increasingly made at a higher level of abstraction. The relevance of models and the efficiency of model-based development have been demonstrated for software engineering. A comparable approach is applicable to quality-assurance activities including testing. The concept of model-based testing is emerging in its application for embedded systems.

Nowadays, 44% of embedded system designs meet only 20% of functionality and performance expectations [Enc03, Hel$^+$05]. This is partially attributed to the lack of an appropriate test approach for functional validation and verification. Hence, the problem addressed by this innovation relates to quality-assurance processes at model level, when neither code nor hardware exists. A systematic, structured, and abstract test specification is in the primary focus of the innovation. In addition, automation of the test process is targeted as it can considerably cut the efforts and cost of development.

The main contribution of this thesis applies to the *software* built into *embedded systems*. In particular, it refers to the *software models* from which *systems* are built. An approach to functional black-box testing based on the system models by providing a test model is developed. It is contrasted with the currently applied test methods that form dedicated solutions, usually specialized in a concrete testing context. The test framework proposed herewith, is realized in the MATLAB®/Simulink®/Stateflow® [MathML, MathSL, MathSF] environment and is called ***Model-in-the-Loop for Embedded System Test (MiLEST)***.

The developed ***signal-feature – oriented paradigm*** allows the abstract description of signals and their properties. It addresses the problem of missing reference signal flows as well as the issue of systematic test data selection. Numerous signal features are identified. Furthermore, predefined test patterns help build ***hierarchical test specifications***, which enables a construction of the test specification along modular divide-and-conquer principles. The processing of both discrete and continuous signals is possible, so that the ***hybrid behavior*** of embedded systems can be addressed.
The testing with MiLEST starts in the requirements phase and goes down to the test execution level. The essential steps in this test process are automated, such as the test data generation and test evaluation to name the most important.

Three case studies based on adaptive cruise control are presented. These examples correspond to component, component-in-the-loop, and integration level tests. Moreover, the quality of the test specification process, the test model, and the resulting test cases is investigated in depth. The resulting test quality metrics are applied during the test design and test execution phases so as to assess whether and how the proposed method is more effective than established techniques. A ***quality gain*** of at least 20% has been estimated.

Zusammenfassung

Die Forschung im Bereich Software-Aspekte von eingebetteten Systemen wird in naher Zukunft entscheidenden Einfluss auf Industrie-, Markt- und Alltagsleben haben. Das regt die Untersuchung dieses Anwendungsgebietes an. Weiterhin wird die Erstellung eines konsistenten, wiederverwendbaren und gut dokumentierten Modells die wichtigste Aufgabe bei der Entwicklung von eingebetteten Systemen. Designentscheidungen, die früher auf der Kodeebene beschlossen wurden, werden heute zunehmend auf einer höheren Abstraktionsebene getroffen. Außerdem, wenn die Debatte über die Relevanz von Modellen und modellbasierter Entwicklung für die Softwaretechnik zutreffend ist, dann besitzt sie auch Gültigkeit für Aktivitäten der Qualitätssicherung einschließlich Testen. Hiermit wird das Konzept des modellbasierten Testens entwickelt.
Heutzutage erfüllen 44% der eingebetteten Systemdesigns 20% der Erwartungen an Funktionalität und Leistung [Enc03, Hel^{+}05]. Das liegt zum Teil daran, dass ein passender Testansatz für funktionale Validierung und Verifikation fehlt. Folglich bezieht sich das in dieser Dissertation besprochene Problem auf den Qualitätssicherungsprozess auf Modellebene, wenn weder Kode noch Hardware existiert. Eine systematische, strukturierte, wiederholbare und möglichst abstrakte Testspezifikation ist der Hauptschwerpunkt dieser Arbeit. Ein weiteres Ziel ist eine Automatisierung des Testprozesses, da diese den Arbeitsaufwand und die Kosten der Entwicklung beträchtlich senken kann.

Der Hauptbeitrag dieser Dissertation gilt für *Software der eingebetteten Systemen* und bezieht sich die eigentliche Breite dieser Arbeit auf *Modelle des Softwares*, auf deren Grundlage folglich die Systeme gebaut werden. Ein Ansatz für funktionale Black-Box Tests, die auf den Modellen basieren und die selbst auch ein Testmodell darstellen, wurde entwickelt. Dem stehen derzeit verwendete Testmethoden gegenüber, die zweckbestimmte Lösungen für in der Regel spezialisierte Testzusammenhänge darstellen. Die hier vorgeschlagene Testframework wurde in einer MATLAB®/Simulink®/Stateflow®-Umgebung realisiert und trägt den Namen ***Model-in-the-Loop for Embedded System Test (MiLEST)***.

Das Signalsmerkmals-orientierte Paradigma erlaubt eine abstrakte Beschreibung eines Signals und spricht sowohl die Probleme des fehlenden Verlaufes von Referenzsignalen als auch der systematischen Testdatenauswahl an. Zahlreiche Signalsmerkmale werden identifiziert und klassifiziert, vordefinierte Testmuster helfen, ***hierarchische Testspezifikationen*** zu bilden. Dadurch wird die Verarbeitung von diskreten und kontinuierlichen Signalen möglich, so dass das ***hybride Verhalten*** des Systems adressiert wird.
Das Testen mittels MiLEST beginnt in der Anforderungsphase und geht hinunter auf das Testdurchführungsniveau. Einige Prozessschritte sind automatisiert, wobei die Testdatengenerierung und die Testauswertung zu den wichtigsten zählen.

Drei Fallstudien, die auf der Funktionalität des Tempomats basieren, werden vorgestellt. Diese Beispiele entsprechen den Komponententests, Component-in-the-Loop-Tests und Integrationsniveautests. Außerdem, werden die Qualität des Testspezifikationsprozesses, des Testmodells und der resultierenden Testfälle genauer untersucht. Die Testqualitätsmetriken werden dann während der Testkonstruktion und der Testdurchführung angewendet, um einzuschätzen, ob und in welchem Maße sich die vorgeschlagene Methode von bekannten Techniken unterscheidet. ***Qualitätsgewinn*** von mindestens 20% wird abgeschätzt.

Declaration

The work presented in this thesis is original work undertaken between September 2005 and August 2008 at the Fraunhofer Institute for Open Communication Systems, *Competence Center – Modeling and Testing for System and Service Solutions*, the Technical University of Berlin, *Faculty of Electrical Engineering and Computer Science*, and the University of California, San Diego, *Department of Computer Science and Engineering*. It has been financed by several research grants and the doctoral fellowship awarded to the author by *Studienstiftung des deutschen Volkes*[1]. Portions of this work have been already presented elsewhere due to a number of research travel scholarships received from *the IFIP*, *IEEE*, *Siemens,* and *Métodos y Tecnología*. They resulted in the following publications:

- ZANDER-NOWICKA, J., SCHIEFERDECKER, I., MARRERO PÉREZ, A.: Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems, In *Proceedings of the IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006)*, Pages: 799-805, IEEE Catalog Number: 06CH37750C, ISBN: 1-4244-0052-X, ISSN: 1088-7725, Anaheim, CA, U.S.A. 2006.

- ZANDER-NOWICKA, J., SCHIEFERDECKER, I., FARKAS, T.: Derivation of Executable Test Models From Embedded System Models using Model Driven Architecture Artefacts - Automotive Domain, In *Proceedings of the Model Based Engineering of Embedded Systems II (MBEES II)*, Editors: Giese, H., Rumpe, B., Schätz, B., TU Braunschweig Report TUBS-SSE 2006-01, Dagstuhl, Germany. 2006.

- ZANDER-NOWICKA, J., MARRERO PÉREZ, A., SCHIEFERDECKER, I.: From Functional Requirements through Test Evaluation Design to Automatic Test Data Retrieval – a Concept for Testing of Software Dedicated for Hybrid Embedded Systems, In *Proceedings of the IEEE 2007 World Congress in Computer Science, Computer Engineering, & Applied Computing; The 2007 International Conference on Software Engineering Research and Practice (SERP 2007)*, Editors: Arabnia, H. R., Reza, H., Volume II, Pages: 347-353, ISBN: 1-60132-019-1, Las Vegas, NV, U.S.A. CSREA Press, 2007.

- ZANDER-NOWICKA, J., MARRERO PÉREZ, A., SCHIEFERDECKER, I., DAI, Z. R.: Test Design Patterns for Embedded Systems, In *Business Process Engineering. Conquest-Tagungsband 2007 – Proceedings of the 10th International Conference on Quality Engineering in Software Technology (CONQUEST 2007)*, Editors: Schieferdecker, I., Goericke, S., ISBN: 3898644898, Potsdam, Germany. Dpunkt.Verlag GmbH, 2007.

- ZANDER-NOWICKA, J.: Reactive Testing and Test Control of Hybrid Embedded Software, In *Proceedings of the 5th Workshop on System Testing and Validation (STV 2007), in conjunction with ICSSEA 2007*, Editors: Garbajosa, J., Boegh, J., Rodriguez-Dapena, P., Rennoch, A., Pages: 45-62, ISBN: 978-3-8167-7475-4, Paris, France. Fraunhofer IRB Verlag, 2007.

[1] Studienstiftung des deutschen Volkes – http://www.studienstiftung.de [04/09/2008].

- ZANDER-NOWICKA, J., XIONG, X., SCHIEFERDECKER, I.: Systematic Test Data Generation for Embedded Software, In *Proceedings of the IEEE 2008 World Congress in Computer Science, Computer Engineering, & Applied Computing; The 2008 International Conference on Software Engineering Research and Practice (SERP 2008)*, Editors: Arabnia, H. R., Reza, H., Volume I, Pages: 164-170, ISBN: 1-60132-086-8, Las Vegas, NV, U.S.A. CSREA Press, 2008.

- ZANDER-NOWICKA, J.: Model-Based Testing of Real-Time Embedded Systems for Automotive Domain, In *Proceedings of the International Symposium on Quality Engineering for Embedded Systems (QEES 2008)*; in conjunction with 4th European Conference on Model Driven Architecture – Foundations and Applications, Pages: 55-58, ISBN: 978-3-8167-7623-9, Berlin. Fraunhofer IRB Verlag, 2008.

- ZANDER-NOWICKA, J., MOSTERMAN, J. P., SCHIEFERDECKER, I.: Quality of Test Specification by Application of Patterns, In *Proceedings of the 2nd International Workshop on Software Patterns and Quality (SPAQu 2008)*; in conjunction with 15th Conference on Pattern Languages of Programs (PLoP 2008), co-located with OOPSLA 2008, Nashville, TN, U.S.A. 2008.

- ZANDER-NOWICKA, J., SCHIEFERDECKER, I.: Model-based Testing. In *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, Editors: Gomes, L., Fernandes, J. M., *to appear*. IGI Global, 2009.

The authors tend to protect their invention in the market. Hence, the proposed algorithms for software quality assurance, the corresponding test methodology, and the resulting test execution framework, called MiLEST have been submitted to the United States Patent and Trademark Office (USPTO) on June 20th, 2008 as a patent application.

Acknowledgements

"What shall I say?
Everything that I could say would fade into insignificance
compared with what my heart feels [...] at this moment."

- Karol Wojtyła – John Paul II

First of all, I would like to offer my deepest thanks to my supervisor, *Professor Dr. Ina Schieferdecker*. She has made this endeavor academically possible, shaped research, and instilled in me the quality to cultivate my own potential. She was one of the first who believed in me. Her knowledge and suggestions have proven to be invaluable and have contributed profoundly to the results presented in this thesis. It has been an exceptional privilege to work with her. I dearly appreciate her personality, her work ethics, and the positive energy that she always emits, as well as how it affects and impacts those around her. These all are rare virtues. I would like to express my sincerest gratitude towards her. Thank you so much, Ina!

For the invaluable opportunity to work as a visiting scholar in a dynamic environment at the University of California in San Diego (UCSD) I would like to express my warmest appreciation to my other supervisor, *Professor Dr. Ingolf Heiko Krüger*. Providing me scientific support and professional advice, he has been a person who embodies characteristics that I can only aim to model myself after. I offer my sincerest gratitude for making me aware of the new perspectives that have found their way into this work, as well as enlightened my mind.

My genuine appreciation goes out to my former students and wonderful co-workers, especially *Abel Marrero Pérez* and *Xuezheng Xiong* for their invaluable, precise, and diligent contribution to this work.

Deep respect I express to my work colleagues at Fraunhofer Institute FOKUS in Berlin. In particular, the discussions with *Dr. George Din*, *Alain Vouffo Feudjio,* and *Andreas Hoffmann* are truly appreciated. Likewise, I wish to thank all of my colleagues from the UCSD for our fruitful research interactions and, in particular, *Dr. Emilia Farcas* for her wonderful charm.

I had the honor of discussing the topics related to this thesis with many skilled people, namely our partners in industry *Dr. Pieter J. Mosterman, Dr. Eckard Bringmann, Dr. Mirko Conrad*, *Jens Hermann,* and *Guy Ndem Collins*. It is my privilege to extend my thanks to them for inspiring me with their interest in my research and their comments that improved the quality of this thesis. Also, my visit to The MathWorks helped orient my scientific fundamentals and shape some interest in my future work.

The members of our doctoral TAV-Junior group are gratefully acknowledged, especially *Jürgen Großmann*, *Dr. Stefan Wappler*, *Abel Marrero Pérez,* and *Stephan Weißleder* as well as the members of the doctoral working group UML-based Testing: *Dr. Zhen Ru Dai*, *Dr. Dehla Sokenou*, *Dr. Dirk Seifert,* and *Mario Friske* for their scientific support.

My sincerest appreciation goes out to my dear friends *Zhen Ru Dai*, *George Din*, *Jürgen Großmann*, *Abel Marrero Pérez*, *Axel Rennoch,* and *Diana Vega* for helping me out on a number of occasions, perhaps without even realizing it.

Furthermore, I wish to thank my family, my relatives, and my friends from Poland I have known for a long time. I would never have finished this work if my parents had not instilled the value of labor and the demeanor of hard work in me. My deepest gratitude goes to my twin sister *Sylwia Fil* for believing that I am more than I think I really am. My warmest thanks are for *Łukasz Nowicki*, whose unconditional love prevented me from losing faith in my professional goals. I will be infinitely grateful forever!

In addition, I wish to express my acknowledgements to my personal tutor *Professor Dr. Eckehard Schöll* and his wife *Viola Schöll* for exposing me to the joy at cultural events that will bear a lasting impression. Last but not least, I wish to thank the *Studienstiftung des deutschen Volkes* for a number of financial research grants enabling me much more than I expected, for teaching me what '*elite'* actually means, and for providing me with ideal education conditions.

The flexible and well-equipped environment at all the workplaces where I could conduct my research contributed to the development of this work, for which I am also deeply grateful.

The involvement of all these exceptional *people* and *everybody* else who contributed to my level of education has been instrumental in accomplishing the work in this thesis. In addition and probably more important, you all have helped me become the person that I am. I thank you!

Justyna Zander-Nowicka
Berlin, Germany

Table of Contents

List of Figures VII

List of Tables XII

– Part I – 1

1 Introduction 2

1.1 Background and Motivation 2

1.1.1 Current Trends for Embedded Systems 2

1.1.2 Relevance of Model-based Activities 3

1.1.3 Quality and Testing 4

1.1.4 Automotive Domain 5

1.2 Scope, Contributions and Structure of the Thesis 5

1.3 Roadmap of the Thesis 8

2 Fundamentals 11

2.1 Yet Another System under Test 11

2.1.1 Embedded System 11

2.1.2 Hybrid System 12

2.1.3 Reactive System 12

2.1.4 Real-Time System 13

2.1.5 Electronic Control Unit in the Automotive 13

2.1.6 Control Theory 14

2.2 Model-based Development 16
2.2.1 Issues in Model-based Development 16
2.2.2 Other Model-based Technologies 17
2.2.3 MATLAB/Simulink/Stateflow as a Framework 18
2.3 Testing 21
2.3.1 Software Testing 21
2.3.2 Test Dimensions 23
2.3.3 Requirements on Embedded Systems Testing within Automotive 28
2.3.4 Definition and Goals of Model-based Testing 29
2.3.5 Patterns 30
2.4 Summary 30

3 Selected Test Approaches 32
3.1 Categories of Model-based Testing 32
3.1.1 Test Generation 34
3.1.2 Test Execution 37
3.1.3 Test Evaluation 38
3.2 Automotive Practice and Trends 40
3.3 Analysis and Comparison of the Selected Test Approaches 41
3.3.1 Analysis of the Academic Achievements 42
3.3.2 Comparison of the Test Approaches Applied in the Industry 43
3.4 Summary 49

– Part II – 51

4 A New Paradigm for Testing Embedded Systems 52
4.1 A Concept of Signal Feature 53
4.1.1 A Signal 53
4.1.2 A Signal Feature 55
4.1.3 Logical Connectives in Relation to Features 56
4.1.4 Temporal Expressions between Features 56
4.2 Signal Generation and Evaluation 58
4.2.1 Features Classification 58
4.2.2 Non-Triggered Features 63
4.2.3 Triggered Features 70
4.2.4 Triggered Features Identifiable with Indeterminate Delay 77
4.3 The Resulting Test Patterns 88
4.4 Test Development Process for the Proposed Approach 89

4.5 Related Work 95
4.5.1 Property of a Signal 95
4.5.2 Test Patterns 95
4.6 Summary 96

5 The Test System 97

5.1 Hierarchical Architecture of the Test System 98
5.1.1 Test Harness Level 99
5.1.2 Test Requirement Level 99
5.1.3 Test Case Level – Validation Function Level 102
5.1.4 Feature Generation Level – Feature Detection Level 104
5.2 Test Specification 112
5.3 Automation of the Test Data Generation 116
5.3.1 Transformation Approach 116
5.3.2 Transformation Rules 117
5.4 Systematic Test Signals Generation and Variants Management 120
5.4.1 Generation of Signal Variants 120
5.4.2 Test Nomenclature 123
5.4.3 Combination Strategies 124
5.4.4 Variants Sequencing 126
5.5 Test Reactiveness and Test Control Specification 128
5.5.1 Test Reactiveness Impact on Test Data Adjustment 129
5.5.2 Test Control and its Relation to the Test Reactiveness 130
5.5.3 Test Control Patterns 132
5.6 Model Integration Level Test 133
5.6.1 Test Specification Design Applying the Interaction Models 133
5.6.2 Test Data Retrieval 135
5.6.3 Test Sequence versus Test Control 136
5.7 Test Execution and Test Report 136
5.8 Related Work 137
5.8.1 Test Specification 137
5.8.2 Transformation Possibilities 137
5.9 Summary 138

– Part III – 140

6 Case Studies 141

6.1 Adaptive Cruise Control 141

6.2 Component Level Test for Pedal Interpretation 144
6.2.1 Test Configuration and Test Harness 145
6.2.2 Test Specification Design 146
6.2.3 Test Data and Test Cases 147
6.2.4 Test Control 149
6.2.5 Test Execution 150
6.3 Component in the Loop Level Test for Speed Controller 156
6.3.1 Test Configuration and Test Harness 157
6.3.2 Test Specification Design 158
6.3.3 Test Data and Test Reactiveness 159
6.3.4 Test Control and Test Reactiveness 161
6.3.5 Test Execution 162
6.4 Adaptive Cruise Control at the Model Integration Level 164
6.4.1 Test Configuration and Test Harness 164
6.4.2 Test Specification Applying Interaction Models 164
6.4.3 Test Specification Design 166
6.4.4 Test Data Derivation 168
6.4.5 Test Control 171
6.4.6 Test Execution 171
6.5 Summary 172

7 Validation and Evaluation 173

7.1 Prototypical Realization 173
7.2 Quality of the Test Specification Process and Test Model 177
7.2.1 Test Quality Criteria 177
7.2.2 Test Quality Metrics 178
7.2.3 Classification of the Test Quality Metrics 182
7.3 The Test Quality Metrics for the Case Studies 184
7.3.1 Pedal Interpretation 185
7.3.2 Speed Controller 185
7.3.3 Adaptive Cruise Control 186
7.3.4 Concluding Remarks 186
7.4 Quality of the Test Strategy 187
7.5 Limitations and Scope of the Proposed Test Method 188
7.6 Summary 189

8 Summary and Outlook 190

8.1 Summary 190

8.2 Outlook 193

8.3 Closing Words 195

Glossary 197

Acronyms 200

Bibliography 206

Appendix A 225

List of Model-based Test Tools and Approaches 225

Appendix B 233

Test Patterns Applicable for Building the Test System 233

Appendix C 235

Hierarchical Architecture of the Test System 236

Appendix D 237

Questionnaire 237

Appendix E 238

Contents of the Implementation 238

Appendix F 241

Index 242

List of Figures

Figure 1.1: Dependencies between Chapters. 8

Figure 1.2: Roadmap for Gaining the General Overview of this Thesis. 9

Figure 1.3: Roadmap for Studying the Proposed Test Methodology. 10

Figure 1.4: Roadmap for Studying the Implementation. 10

Figure 2.1: Interactions of Embedded System with the Environment. 12

Figure 2.2: A Simple Closed–Loop Controller. 15

Figure 2.3: The Multiple V-Model. 16

Figure 2.4: The Five Test Dimensions. 23

Figure 3.1: Overview of the Taxonomy for Model-based Testing. 34

Figure 3.2: Trapezoid Selecting the Range of the Test Approaches. 43

Figure 4.1: A Hybrid Embedded System with Discrete- and Continuous-timed Signals. 55

Figure 4.2: A Descriptive Approach to Signal Feature. 55

Figure 4.3: Signal-Features Generation – a few instances. 59

Figure 4.4: Signal-Features Generation – a Generic Pattern. 59

Figure 4.5: Signal-Features Evaluation – a Generic Pattern. 60

Figure 4.6: Signal-Features Classification based on the Feature Identification Mechanism. 62

Figure 4.7: Feature Generation: Increase Generation and the Corresponding SL Block masks. 66

Figure 4.8: Feature Extraction: Increase. 67

Figure 4.9: Feature Extraction: Constant. 67

Figure 4.10: Relative Tolerance Block. 68

Figure 4.11: Feature Generation: Linear Functional Relation. 68

Figure 4.12: Feature Extraction: Linear Functional Relation. 68

Figure 4.13: Feature Extraction: Minimum to Date. 69

Figure 4.14: Feature Extraction: Local Maximum. 69

Figure 4.15: Feature Extraction: Peak. 70

Figure 4.16: Feature Generation: Time Stamp of an Event. 73

Figure 4.17: Feature Extraction: Time Stamp of an Event. 74

Figure 4.18: Feature Generation: Time since Signal is Constant. 74

Figure 4.19: Feature Extraction: Time since Signal is Constant. 75

Figure 4.20: Feature Extraction: Signal Value at Maximum. 75

Figure 4.21: Examples for Generation of TDD Features Related to Maximum and their Corresponding SL Block Masks:
Time Stamp at Maximum = Value,
Time Stamp at Maximum with given Slope,
Time Stamp at Maximum,
Signal Value at Maximum. 76

Figure 4.22: Feature Generation: Time between Two Events. 79

Figure 4.23: Feature Generation: Signal Mean Value in the Interval between Two Events. 79

Figure 4.24: Reaction on a Step Function:
a) A Step Function – u(kT).
b) Step Response Characteristics y(kT): rise time (t_r), maximum overshoot, settling time (t_s) and steady-state error (e_{ss}). 81

Figure 4.25: Step Response of Stable Second-Order System for Different Damping Ratios. 83

Figure 4.26: Reset Signal Extraction for Step Response Evaluation. 83

Figure 4.27: Constancy Check for a Given Minimal Time within the Reset Signal Extraction.84

Figure 4.28: Step Detection within the Reset Signal Extraction. 84

Figure 4.29: Feature Signals Extraction for Step Response Evaluation. 85

Figure 4.30: Computing Response Step Size, Step Size and Expected Set Point. 86

Figure 4.31: Feature Extraction: Rise Time. 86

Figure 4.32: Time Difference Block for Rise Time Detection. 87

Figure 4.33: Feature Extraction: Settling Time. 87

Figure 4.34: Feature Extraction: Steady-State Error. 87

Figure 4.35: Extraction of Trigger Signals. 88

Figure 4.36: Test Development Process. 90
Figure 4.37: A Test Harness Pattern. 91
Figure 4.38: A Pattern for the Test Requirement Specification. 92
Figure 4.39: Structure of a VF – a Pattern and its GUI. 93
Figure 4.40: Assertion Block – a Pattern. 93
Figure 4.41: Test Requirement Level within the Test Data Unit – a Pattern. 93
Figure 4.42: Structure of the Test Data Set – a Pattern. 94

Figure 5.1: Hierarchical Architecture of the Test System. 98
Figure 5.2: Fundamental Structure of the Test Requirement Level – TDGen View. 100
Figure 5.3: Fundamental Structure of the Test Requirement Level – TSpec View. 100
Figure 5.4: Arbitration Mechanism. 101
Figure 5.5: Fundamental Structure of the Test Case Level. 102
Figure 5.6: Exemplified Structure of the Test Case Level. 103
Figure 5.7: Fundamental Structure of the Validation Function Level. 104
Figure 5.8: Fundamental Structure of the Feature Generation Level. 105
Figure 5.9: Variants Management Structure. 105
Figure 5.10: Conversions of the identification mechanisms for TI – TDD and TDD – TID features. 106
Figure 5.11: Fundamental Structure of the Feature Detection Level for Preconditions. 107
Figure 5.12: Preconditions Synchronization Parameter Mask. 107
Figure 5.13: Preconditions Synchronization – an Example. 109
Figure 5.14: Fundamental Structure of the Feature Detection Level for Assertions. 111
Figure 5.15: Implementation of during(x) TI feature – Feature Detection Level (Assertions).115
Figure 5.16: Insights of the SF Diagram for the Implementation of during(x) TI feature. 115
Figure 5.17: Retarding the Extraction of TI Feature by Application of after(y). 116
Figure 5.18: Test Stimuli Definition – an Abstract View. 119
Figure 5.19: Test Stimuli Definition – a Concrete View. 119
Figure 5.20: Steps of Computing the Representatives. 121
Figure 5.21: Steps of Computing the Representatives for Vehicle Velocity. 123
Figure 5.22: Minimal Combination. 125
Figure 5.23: One Factor at a Time Combination. 125
Figure 5.24: Pair-wise Combination. 126

Figure 5.25: Test Control and its Implication on the Test Data Sequencing. 128
Figure 5.26: Variant-Dependent Test Control. 132
Figure 5.27: Test Control Insights for Four Test Cases. 132
Figure 5.28: Basic hySCt. 134
Figure 5.29: HySCt including VFs Concepts. 135
Figure 5.30: The Metamodel Hierarchy of MiLEST. 138

Figure 6.1: Components of the ACC System. 143
Figure 6.2: Components of the Cruise Control Subsystem. 143
Figure 6.3: Pedal Interpretation Insights. 145
Figure 6.4: The Test Harness around the Pedal Interpretation. 145
Figure 6.5: Test Specification for Requirement 2.2. 146
Figure 6.6: Preconditions Set: v = const & phi_Acc increases & T_des_Drive >= 0. 147
Figure 6.7: Assertion: T_des_Drive increases. 147
Figure 6.8: Derived Data Generators for Testing Requirement 2.2. 148
Figure 6.9: Test Data for one Selected Precondition Set. 148
Figure 6.10: Parameterized GUIs of Increase Generation. 149
Figure 6.11: Test Control for Ordering the Test Cases Applying Minimal Combination Strategy. 150
Figure 6.12: Execution of Test Case 4 Applying the 4th Test Data Variants Combination. 153
Figure 6.13: Resulting Test Data Constituting the Test Cases According to Minimal Combination Strategy. 154
Figure 6.14: SUT Outputs for the Applied Test Data (1). 155
Figure 6.15: SUT Outputs for the Applied Test Data (2). 156
Figure 6.16: The Speed Controller Connected to a Vehicle Model via a Feedback Loop. 157
Figure 6.17: A Test Harness for the Speed Controller. 158
Figure 6.18: Test Requirements Level within the Test Specification Unit. 158
Figure 6.19: VFs within the Test Specification for the Speed Controller. 159
Figure 6.20: Test Requirements Level within the Test Data Generation Unit. 160
Figure 6.21: Test Data Set for Test Case 1. 160
Figure 6.22: Influence of Trigger Value on the Behavior of Test Data. 161
Figure 6.23: Influence of Verdict Value from Test Case 1 on the Test Control. 162
Figure 6.24: Results from the Test Execution of the Speed Controller. 163

Figure 6.25: Test Harness for ACC. 164
Figure 6.26: ACC Flow – Possible Interactions between Services. 165
Figure 6.27: Service: Braking when ACC System is Active Trigger. 166
Figure 6.28: A Set of VFs Retrieved from the hySCts. 167
Figure 6.29: Preconditions for the VF: Braking when ACC active_trigger. 168
Figure 6.30: Assertions for the VF: Braking when ACC active_trigger. 168
Figure 6.31: Test Data Set for a Selected ACC Model Integration Test. 169
Figure 6.32: Test Control for a Selected ACC Model Integration Test. 171
Figure 6.33: The Car Velocity Being Adjusted. 172

Figure 7.1: Integration of MiLEST in the MATLAB/Simulink environment. 174
Figure 7.2: Overview of the MiLEST Library. 174
Figure 7.3: Overview of MiLEST Library. 176
Figure 7.4: Implementation of the VFs Activation Coverage Exemplified for Two of Them. 181

Figure 8.1: Overview of the Thesis Contents. 192

List of Tables

Table 3.1: Classification of the Selected Test Approaches based on the MBT Taxonomy. 44

Table 3.2: Classification of the Selected Test Approaches based on the Test Dimensions. 46

Table 3.3: Classification of the Test Approaches based on the Selected Criteria. 48

Table 3.4: Classification of MiLEST with respect to the MBT Taxonomy. 49

Table 4.1: TI Features – Evaluation and Generation Algorithms. 64

Table 4.2: TDD Features – Evaluation and Generation Algorithms. 72

Table 4.3: TID Features – Evaluation and Generation Algorithms. 78

Table 4.4: Illustration of Reasoning about Patterns. 95

Table 5.1: Transformation Rules for Test Data Sets Retrieval. 118

Table 5.2: Options for Increase Generation. 122

Table 5.3: Test Data Generation Dependencies. 129

Table 5.4: Test Control Principles on the Component Test Level. 131

Table 6.1: Selected Requirements for Adaptive Cruise Control. 142

Table 6.2: Requirements for Pedal Interpretation (excerpt). 144

Table 6.3: SUT Inputs of Pedal Interpretation Component. 144

Table 6.4: SUT Outputs of Pedal Interpretation Component. 145

Table 6.5: SUT Inputs of Speed Controller. 157

Table 6.6: Requirements on Speed Controller. 158

Table 6.7: Design Principles Used to Support the Test Reactiveness in the Speed Controller Test. 161

Table 6.8: Relation of Test Data Sets to Services. 170

Table 7.1: Classification of MiLEST Test Quality Metrics. 183

Table 7.2: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the Pedal Interpretation. 185

Table 7.3: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the Speed Controller. 185

Table 7.4: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the ACC. 186

– Part I –
General

1 Introduction

"Science arose from poetry... when times change
the two can meet again on a higher level
as friends."

- Johann Wolfgang von Goethe

1.1 Background and Motivation

The worldwide market for advanced embedded controllers is growing strongly, driven mainly by the increasing electronic applications in vehicles and the need for comfort and convenience. Studies in [FS08] expect the European market to grow to € 1.14 billion in 2015 at a compound annual growth rate (CAGR) of 10.9% from € 499 million in 2007, which substantiates the growth of the advanced electronic control unit (ECU) market. Giving the background for comparison, the worldwide hardware and software market is expected to grow by 8% per annum [Kri05], whereas the average annual growth rate of the gross domestic product (GDP) has been 5% between 2004 and 2006 [OECD08]. Global light-vehicle production is forecast to grow from 67 million in 2007 to 80 million in 2015 [AES08].

Furthermore, software shows the highest growth rate within embedded systems. The estimated average annual growth rates between 2004 and 2009 are 16% for embedded software [Kri05, Hel$^+$05, OECD05].

Within the past few years the share of software-controlled innovations in the automotive industry has increased from 20% to 85% [Hel$^+$05] and is still growing [MW04, SZ06, BKP$^+$07]. Studies predict that software will determine more than 90% [KHJ07] of the functionality of automotive systems in the near future. Consequently the impact of software on the customer and, hence, on market shares and competition will be enormous. [LK08] conclude that software is established as a key technology in the automotive domain.

1.1.1 Current Trends for Embedded Systems

This thesis is primarily focused on the research on the software aspects of embedded systems, since this field is expected to have the greatest impact on industry, market, and everyday life in the near future [BKP$^+$07]. The increasing system functionality can only be realized by a reasonable shift from hardware to software [LK08]. Software development offers more flexibility,

more variants can be built and, finally, development time and cost can be reduced. At the same time, this software often plays a critical role and a failure can result in expensive damage to equipment, injury, or even death.
Moreover, the widening design productivity gap has to be addressed by advances in design capabilities. Automation of the process is greatly needed. This applies not only to embedded systems development and deployment, but also, consequently, to quality assurance of the entire development process, and the resulting software – not to mention the fundamental research in any of these areas [Hel$^+$05].

1.1.2 Relevance of Model-based Activities

The creation of consistent, reusable, and well-documented models becomes the important stage in the development of embedded systems [Hel$^+$05]. Hence, the concept of model-based development (MBD) emerges. Due to the increasing complexity of the developed systems it is necessary to model correctly and to implement the chosen design in a correct manner. The future importance of design-level methods and tools is illustrated by the current shift from implementation towards design [UL06, CD06]. A lot of decisions formerly made during the implementation phase should already be done on a higher level of abstraction. The paradigm shift is also reflected by the increasing use of modeling tools, such as MATLAB®/Simulink®/Stateflow® (ML/SL/SF) [MathML, MathSL, MathSF] or UML®-based tools [UML, BBH04, BGD07].

With the trend towards behavioral modeling in several embedded systems domains, the implementation of such models becomes gradually more straightforward. This implies a new standard of the engineers' competence at the design level [Hel$^+$05]. Subsequently, the role of the traditional programmer becomes more restricted since the properly designed models are often executable. Virtual prototyping, integration, or transformation into different kinds of model is already possible. Hence, substantial parts of the implementation are generated based on models. The generated code, if optimized[2], is compiled in the traditional way for different platforms. Such an approach is manifested within the standardization efforts of the Object Management Group (OMG) in the context of Model Driven Architecture (MDA) [MDA], where platform-independent models (PIMs) can be enriched with platform-specific information. The resulting platform specific models (PSMs) are then the basis for code generation. Also, AUTomotive Open System Architecture (AUTOSAR) [ZS07] contains the idea of MDA, although a bit more specific. It shows that in the near future, generation of the optimized code from models will be possible without losing the precision of handwritten code.

MBD introduction is clearly to be observed in the automotive domain. At the end of the 1990s a paradigm shift in the development of software-based vehicle functions was initiated. Traditional software development on the basis of textual specifications and manual coding has not been possible any more due to the increasing complexity of software-intensive systems, especially in the context of control theory. Hence, MBD emerged with the application of executable

[2] The limited resources of the embedded systems for which the code is generated require optimization techniques to be applied whenever possible in order to generate efficient code with respect to different dimensions (e.g., memory consumption or execution speed) [SCD$^+$07].

models and automatic coding [SM01, CFG$^+$05, LK08] in its background. Within MBD, an executable functional model of the ECU is created at an early stage in the development process. The model consists usually of block diagrams and extended state machines. It can be simulated together with a plant (e.g., a vehicle) so as to be implemented on the ECU afterwards. Due to the availability of executable models, analytical methods can be applied early and integrated into subsequent development steps. The models form a basis for further activities, such as code generation, model transformations, validation, verification, or testing. The positive effects such as early error detection and early bug fixing are obvious [CFB04, SG07].

In this thesis, the principles of system development apply to the test system as well. If the discussion about the relevance of models and model-based development is true for software and system production, it is also valid for their quality-assurance activities, obviously, including testing [Dai06, ISTQB06]. With this practice, the concept of model-based testing emerges.

1.1.3 Quality and Testing

An *embedded system* [BBK98, LV04] is a system built for dedicated control functions. Unlike standard computer systems, embedded systems do not usually come with peripheral devices, since hardware is minimized to the greatest possible extent. *Embedded software* [LV04] is the software running on an embedded system. Embedded systems have become increasingly sophisticated and their software content has grown rapidly in the last few years. Applications now consist of hundreds of thousands, or even more, lines of code. The requirements that must be fulfilled while developing embedded software are complex in comparison to standard software. Embedded systems are often produced in large volumes and the software is difficult to update once the product is deployed. Embedded systems interact with real-life environment. Hybrid aspects are often expressed via mathematical formulas. In terms of software development, increased complexity of products, shortened development cycles, and higher customer expectations of quality implicate the extreme importance of software testing. Software development activities in every phase are error prone, so the process of defect detection plays a crucial role. The cost of finding and fixing defects grows exponentially in the development cycle. The software testing problem is complex because of the large number of possible scenarios. The typical testing process is a human-intensive activity and as such it is usually unproductive and often inadequately done. Nowadays, testing is one of the weakest points of current development practices. According to the study in [Enc03] 50% of embedded systems development projects are months behind schedule and only 44% of designs meet 20% of functionality and performance expectations. This happens despite the fact that approximately 50% of total development effort is spent on testing [Enc03, Hel$^+$05]. The impact of research on test methodologies that reduce this effort is therefore very high and strongly desirable [ART05, Hel$^+$05].

Although, a number of valuable efforts in the context of testing already exist, there is still a lot of space to improve the situation. This applies in particular to the automation potential of the test methods. Also, a systematic, appropriately structured, repeatable, and consistent test specification is still an aim to be reached. Furthermore, both abstract and concrete views should be supported so as to improve the readability, on the one hand, and assure the executability of the resulting test, on the other. In the context of this work, further factors become crucial. The testing method should address all aspects of a tested system – whether a mix of discrete and continuous signals, time-constrained functionality, or a complex configuration is considered. In order to establish a controlled and stable testing process with respect to time, budget and software quality, the software testing process must be modeled, measured and analyzed [LV04].

The existence of executable system models opens the potential for model-based testing (MBT). Nowadays, MBT is widely used; however, with slightly different meanings. In the automotive industry MBT is applied to describe all testing activities in the context of MBD [CFS04, LK08]. It relates to a process of test generation based on the model of a system under test (SUT). A number of sophisticated methods representing the automation of black-box test design [UL06] are used. Surveys on different MBT approaches are given in [BJK$^+$05, Utt05, UL06, UPL06, D-Mint08]. This will be discussed in Section 2.3.4 in detail.

In this thesis, additionally, requirements-based testing is considered. Furthermore, a graphical form of a test design will increase the readability. The provided test patterns will considerably reduce the test specifications effort and support their reusability. Then, an abstract and common manner of describing both discrete and continuous signals will result in automated test signals generation and their evaluation.

1.1.4 Automotive Domain

Studies show that the strongest impact of embedded systems on the market has to be expected in the automotive industry. The share of innovative electronics and software in the total value of an automobile is currently estimated to be at least 25%, with an expected increase to 40% in 2010 and up to 50% after 2010 [Hel$^+$05, SZ06]. Prominent examples of such electronic systems are safety facilities, advanced driver assistance systems (ADAS), or adaptive cruise control (ACC). These functionalities are realized by software within ECUs. A modern car has up to 80 ECUs [BBK98, SZ06].

Furthermore, the complexity of car software dramatically increases as it implements formerly mechanically or electronically integrated functions. Yet, the functions are distributed over several ECUs interacting with each other.

At the same time, there is a demand to shorten time-to-market for a car by making its software components reliable and safe. Additionally, studies in [Dess01] show that the cost of recalling a car model with a safety-critical failure can be more than the cost of thorough testing/verification. Under these circumstances the introduction of quality-assurance techniques in the automotive domain becomes obvious and will be followed within this work.

1.2 Scope, Contributions and Structure of the Thesis

After a short motivation on the topic discussed in this thesis, the concrete problems handled here, are outlined. Then, the structure of the thesis is provided, followed by its roadmap for reading purposes.

In addition, before the main contributions are explained, a brief report on the scope is given. The major results achieved in this thesis apply to the *software part of embedded systems*, in general. Following the current trends of model-based development, the actual scope of this work refers to the *models of software*, on which, the *systems* are built. To avoid repeating this term*,* whenever *system* (or *system model*, *system design*, *software*) is referred to in the thesis, the *model of a software-intensive embedded system* is usually meant, unless the context is explicitly indicated. This form of reasoning reflects the tendency to study the abstract level of systems within the considered domain [Pre03b, Con04b, Utt05, BFM$^+$05, CFG$^+$05, AKR$^+$06, BDG07]. This practice also reflects the trend that the embedded systems are often seen in a ho-

listic way, i.e., both software and its surrounding hardware trigger the expected functionality [KHJ07].
The main research problem this thesis is concerned with, relates to *assuring the quality of the embedded system by means of testing at the earliest level of its development*. Based on the analysis of the overall software and test development process, the following questions arise:

1. What is the role of a *system model* in relation to quality assurance? What is the role of a *test model* and what elements does such a test model include? What does *MBT* mean in the context of embedded systems? Is it possible to use a *common language* for both system and test specifications?
2. How can *discrete and continuous signals* be handled at the same time? How should a *test framework* be designed and a *test system* realized? What are the reasons and consequences of the design decisions in terms of test generation and test evaluation?
3. How can the process of test specification and test execution *be automated* to the highest possible extent? What is / is not possible to be automated and why?
4. How can *the test quality* of the test method *be assured* itself? Which means should be used and what do they mean in practice?

The resulting contributions of this thesis can be divided into four main areas:

1. *Model-based test methodology* for testing the functional behavior of embedded, hybrid, real-time systems based on the current software development trends from *practice*;
2. In the *scope of this methodology*, a manner to test the behavior of *hybrid systems*, including the algorithms for *systematic test signal generation* and *signal evaluation*;
3. *Synthesis of a test environment* so as to *automate* the creation of a *comprehensive test system*, which is achieved by means of *test patterns* application that are organized into a hierarchy on different abstraction levels;
4. Assurance of the *quality of the resulting test* by providing the *test metrics* and supporting high *coverage* with respect to *different test aspects*.

These are denominated as *challenges* in the following and the *discourses* are tackled for each of them separately, but not in isolation.

For the first *challenge*, now an introductory remark should already be given. The test framework resulting from this thesis is called *Model-in-the-Loop for Embedded System Test (MiLEST)*. It is realized in the ML/SL/SF since currently about 50% [Hel$^+$05] of functional behavior for embedded systems, particularly in the automotive domain, is modeled using this environment. Considering the fact that nowadays the integration of validation, verification, and testing techniques into common design tools is targeted [Hel$^+$05], the argumentation for choosing this framework for test extensions becomes clear. This practice enables to find a common understanding of software quality problems for both system and test engineers.

In order to clarify and solve the *challenges* listed above, in the upcoming paragraphs the structure of this work will be provided with a special emphasis on the given *challenges* and developed contributions for each of them.

This thesis is organized as follows. This chapter gives an overview and scope of the research topics of this thesis. It introduces the problems that the work is dealing with, its objectives, contributions, structure, and roadmap.

Next, Chapter 2 (cf. Figure 1.1) includes the backgrounds on embedded systems, control theory, ML/SL/SF environment, and test engineering. Additionally, the test dimensions are extracted so as to guide the general aims of this work. The emphasis is put on functional, abstract, executable, and reactive tests at the Model-in-the-Loop (MiL) level.

Chapter 3 introduces the related work on MBT with respect to embedded systems. For that purpose an MBT taxonomy is provided. Herewith, a link to the *first challenge* is done. The roles of the system model and test model are analyzed. Also, common language for both of them is applied. The discussion results in a shape of the test methodology proposed in this thesis. It is called Model-in-the-Loop for Embedded System Test (abbreviated as MiLEST) and realized as an ML/SL/SF library.
All of the chapters named so far constitute the *first general part* of the thesis.

The *second part* relates to the test approach developed herewith. Chapter 4 characterizes a new means for signal description by application of signal features. By that, it relates to the *second challenge* answering the question of how to handle continuous and discrete signals simultaneously. The algorithms for signal-feature generation and evaluation are presented. They are used along a nested architecture for the resulting test system, which is described in Chapter 5 in detail. Additionally, an overview of the proposed test development process and its automation is provided. A discussion on test patterns is given so as to support a fast and efficient reusability of the created test specifications. By that, the *third challenge* is addressed.

Chapter 5 utilizes the results of Chapter 4 and addresses the further questions of the *second challenge*. Different abstraction levels of the MiLEST test system are outlined. The test harness level including the patterns for test specification, test data generator, and test control is described. Then, the test requirements level, test case, and validation function levels follow subsequently. Based on that, algorithms for an automatic test stimuli generation are investigated. This relates again to the *third challenge*. The obtained test models can be used for both validating the system models and testing its implementation.
Chapter 5 also includes the first considerations on the integration level tests. Here, benefits of applying different views on the test specification are discussed.

Finally, the *last part* of this thesis reveals the practical substance of the work. In Chapter 6, three case studies are discussed to validate each of the presented concepts in practice. The examples are related to the functionality of an adaptive cruise control utilized in a vehicle.

Afterwards, they are evaluated in Chapter 7, which deals with the test quality metrics for the proposed test methodology, obtained test designs, and generated test cases. This piece of work relates to the *fourth challenge*. The concepts of test completeness, consistency, and correctness are handled herewith.

Chapter 8 completes this work with a summary and outlook. The MiLEST capabilities and limitations are reviewed, the general trends of the quality assurance for embedded systems are recalled and influences of the contributions of this thesis are outlined.

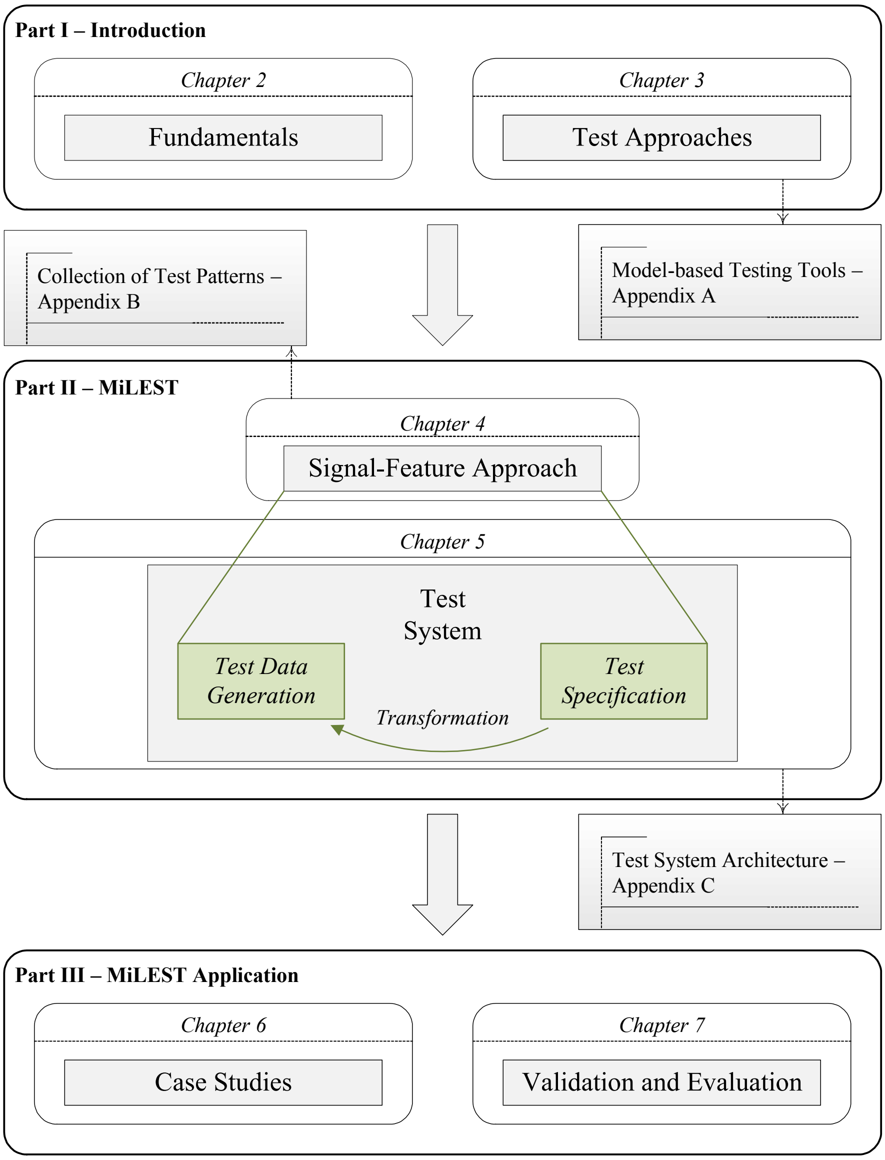


Figure 1.1: Dependencies between Chapters.

1.3 Roadmap of the Thesis

In this section, the dependencies between chapters throughout this thesis are outlined (cf. Figure 1.1). These are also revealed following different reading-paths for this thesis (cf. Figures 1.2 to 1.4).

The detailed discussion on software and test engineering in Chapter 2 serves mostly as a foundation for the considered topics.
Chapter 3 includes a review of the model-based testing approaches. Readers who are familiar with the related work on testing embedded systems are invited to skip this chapter and to consider its contents as additional information.
Chapters 4 and 5 contain the central achievements of the presented work. The concept of signal feature which is presented in Chapter 4 is used as the basis for Chapter 5. Thus, Chapter 5 should not be read without the understanding of the backgrounds outlined in Chapter 4.
Chapters 6 and 7 validate and evaluate the proposed concepts.

The three major perspectives that might be interesting for the reader yield three different reading-paths through this thesis, besides the usual sequential one. Below, a brief survey of these paths is given.

Whenever *an overview* of the subjects and results of the thesis is needed, it is recommended to focus on this introductory chapter, the signal-feature concept discussed in Section 4.1, the evaluation of the soundness and completeness of the contributions described in Chapter 7, and the overall conclusions provided in Chapter 8. Additionally, the reader might want to refer to the summaries given at the end of each chapter, which provide the essential information on their contents.

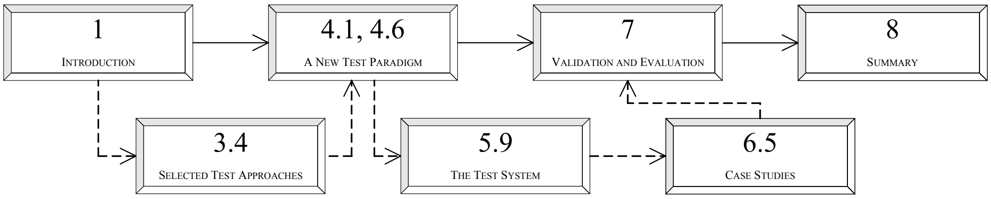


Figure 1.2: Roadmap for Gaining the General Overview of this Thesis.

The profound understanding of the proposed *test methodology* supported by MiLEST may be best instilled through reading selected sections of Chapters 3 – 5 and Chapter 6. To understand the different aspects contributing to the overall shape of the methodology, it is proposed to review the test dimensions (cf. Sections 2.3.2 and 2.3.4) and test categories of the MBT (cf. Section 3.3). A short explanation of the main concept, on which MiLEST is based, can be extracted from Section 4.1. Then, the test development process on an abstract level is introduced in Section 4.4. Its application is revealed in the analysis of the case studies in Chapter 6. For background information on the quality and completeness of the proposed methodology the reader is additionally referred to Chapter 7.

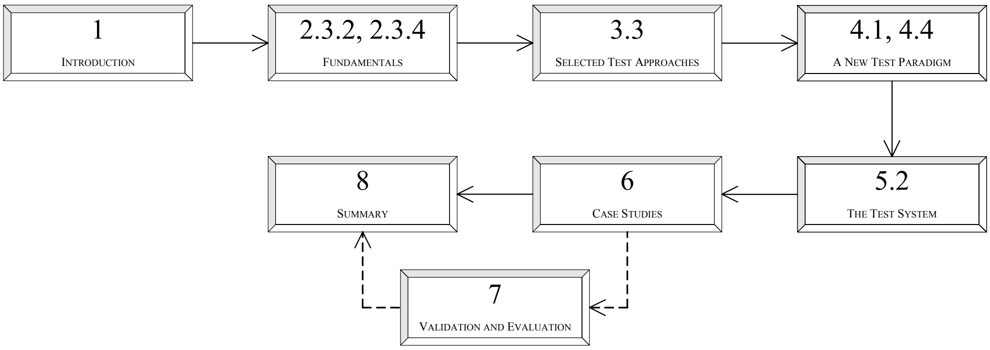


Figure 1.3: Roadmap for Studying the Proposed Test Methodology.

Readers interested in the *MiLEST implementation* that realizes the contributions of this thesis are invited to go through selected sections of Chapter 4, first. There, signal generation and signal evaluation algorithms (cf. Sections 4.1 and 4.2) are provided, both based on the signal-feature concept. To fit the implementation into the entire test development process along MiLEST methodology, Section 4.4 could be helpful. Following this, Chapter 5 includes the detailed technical description of the MiLEST test system and its hierarchical structure.

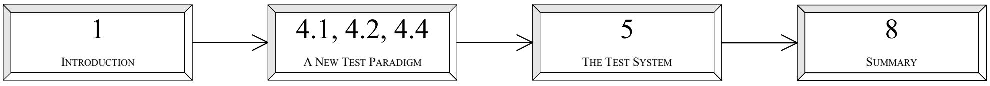


Figure 1.4: Roadmap for Studying the Implementation.

2 Fundamentals

"The important thing is not to stop questioning."

- Albert Einstein

In this chapter the fundamentals of embedded systems, their development and testing are provided. Firstly, in Section 2.1, the notion of the system under consideration from different perspectives is given. Thus, the definitions of embedded, hybrid, reactive, and real-time systems are explained, respectively. In addition, the backgrounds on electronic control units and control theory are discussed. Understanding the characteristics of the *system under test* (SUT) enables further considerations on its development and quality-assurance methods. Further on, in Section 2.2, the concepts of model-based development from the automotive viewpoint are introduced. MATLAB/Simulink/Stateflow (ML/SL/SF) as an example of a model-based development framework is introduced and related approaches are listed.
Section 2.3 gives an insight into the testing world. First, the testing aspects important for the automotive domain are described in detail by categorizing them into different dimensions. Then, requirements on testing within the considered domain are specified. Furthermore, a model-based testing definition and its goals are introduced. A discussion on test patterns completes the theoretical basics.

2.1 Yet Another System under Test

As already discussed in Chapter 1, the main contribution of this work applies to *models of software-intensive embedded systems*. Whenever a *system* (or *system model*, *system design*, *software*) is referred to in this thesis, the *model* is usually meant.

2.1.1 Embedded System

An embedded system (ES) is any computer system or computing device that performs a dedicated function or is designed for use with a specific software application [BBK98, Hel+05]. Instead of dealing with data files, it deals with the control of physical phenomena. It is frequently connected to a physical *environment* through *sensors* and *actuators* as shown in Figure 2.1.

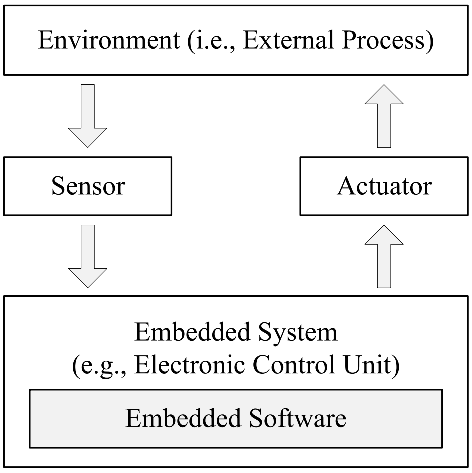

Environment (i.e., External Process)
Sensor
Actuator
Embedded System
(e.g., Electronic Control Unit)
Embedded Software

Figure 2.1: Interactions of Embedded System with the Environment.

ES [Hel+05] is a specialized computer system that is part of a larger system or machine. All appliances that have a digital interface – watches, microwaves, cars, planes, industrial robots, security systems – utilize ESs. Some of them include an operating system, but many are so specialized that the entire logic can be implemented as a single program. Nowadays, software integrated with the ES, also called *embedded software* [Con04b], makes up 85% [Hel+05] of the value of the entire ES.

2.1.2 Hybrid System

Hybrid means generally combining two different technologies or systems. A hybrid system is a dynamic system that exhibits both continuous and discrete dynamic behavior [Hen00]. Its behavior can be described by both differential equations and difference equations. Hybrid systems evolve in continuous time with discrete jumps at particular time instances [Tiw02]. For example, an automobile engine whose continuous fuel injection is regulated by a discrete microprocessor is a hybrid system [Hen00].

2.1.3 Reactive System

In opposite to the transformative systems, reactive systems with a typically non-terminating behavior interact with their environment. As a result, once started, a reactive system operates continually [Krü00]. It accepts input from its environment, it changes its internal state at the same time [Hel+05] and produces corresponding outputs. Reactive systems never halt, although the output of these systems may always be empty from a certain point in time onward.

A reactive system is characterized by a control program that interacts with the environment or another control program [MW91, HP85]. ESs are usually reactive.

2.1.4 Real-Time System

According to [LL90], a real-time system is a computing system where initiation and termination of activities must meet specified timing constraints. The correctness of a computation not only depends on the logical correctness of the system, but also on the time at which the result is produced. A real-time system has to obey hard, soft, and statistical [Dess01, Hel$^+$05] real-time properties.

Hard real-time properties are timing constraints which have to be fulfilled in any case. An example is the autopilot of an aircraft, where violation of hard real-time constraints might lead to a crash.

Soft real-time properties are time constraints which need to be satisfied only in the average case, to a certain percentage, or *fast enough*. An example is video transmission where a delayed frame might either be displayed or dropped, which is not perceivable as long as no consecutive frames are affected [Dess01]. Another example is the software that maintains and updates the flight plans for commercial airliners.

In statistical real time, deadlines may be missed, as long as they are compensated by faster performance elsewhere to ensure that the average performance meets a hard real-time constraint. To be able to fully assess the consequences of the statistical behavior, stochastic analysis is required. However, it is always possible to transform this into a deterministic analysis by investigating the *worst case* situation [Dess01].

Typical examples of timing constraints are:

- the value from the sensor must be read every 100 ms
- the *Worst-Case Execution Time* (WCET) of process A is 160 ms
- it is expected that when event B finishes, event A appears after 10 ms
- it is expected that when event B appears, then during 20 ms signal A will be continuously sent
- it is expected that within 5 ms all events stop

2.1.5 Electronic Control Unit in the Automotive

In the automotive domain, an embedded system is called an electronic control unit (ECU). It controls one or more of the electrical subsystems in a vehicle. In a car, ECUs are connected via bus systems such as, e.g., CAN[3], LIN[4], MOST[5], FlexRay™[6] [Sch06] among others.

[3] CAN in Automation – www.can-cia.org [04/04/08].

[4] LIN Consortium – www.lin-subbus.de [04/04/08].

[5] MOST Cooperation – www.mostnet.org [04/04/08].

[6] FlexRay Group – www.flexray.com [04/04/08].

An instance of embedded system in the form of an ECU controller has been depicted in Figure 2.1. *The external process* is a process that can be of physical, mechanical, or electrical nature [GR06]. *Sensors* provide information about the current state of the *external process* by means of so-called *monitoring events*. They are transferred to the *controller* as input events.
The controller must react to each received input event. Events usually originate from *sensors*. Depending on the received events from sensors, corresponding states of the *external process* are determined.
Actuators receive the results determined by the *controller* which are transferred to the *external process*.

A classification of the application fields of ECUs according to [SZ06] is given below:

- Body (e.g., for headlights, brake lights, air conditioning, power windows)
- Comfort (e.g., for seat and steering-position adjustment, seat heating)
- Engine and power train (e.g., for fuel injection, battery recharging)
- Dashboard for speedometer, odometer, fuel gauge
- Chassis, driving functions
- Telematics and entertainment for audio/video systems

Automotive software, similarly to embedded software for an embedded system, the software is driving an ECU within automobiles.

2.1.6 Control Theory

Considering the ECUs (in particular, closed-loop ECUs), it is inevitable to introduce the basics of control theory. Aristotle[7] [Ack81] already started to think about the control theory [Ben79]. Following his statement *"... if every instrument could accomplish its own work, obeying or anticipating the will of others ... chief workmen would not want servants, nor masters slaves"* [AR], his wish was to automatize the behavior of others (e.g., people, devices) using a set of clearly defined criteria. This is also the idea behind the development of embedded systems – to force them so that they work in a manner they are designed. If so, quality assurance for this development gains the priority too.

Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamic systems. The desired output of a system is called the *reference*. When one or more output variables of a system need to follow a certain reference over time, a controller manipulates the inputs to a system to obtain the desired effect on the output of the system.
Control theory introduces a *feedback*. *Feedback* is a process whereby some proportion of the output signal of a system is passed (fed back) to the input. This is often used to control the dynamic behavior of the system [SG03, MSF05]. A closed-loop controller uses feedback to control states or outputs of a dynamic system. Its name is derived from the information path in the system: process inputs (e.g., voltage applied to an electric motor) have an effect on the process

[7] Aristotle (384 B.C. – 322 B.C.) was a Greek philosopher, a student of Plato and teacher of Alexander the Great. He wrote on many different subjects, including physics, metaphysics, poetry, theater, music, logic, rhetoric, politics, government, ethics, biology and zoology [Ack81].

outputs (e.g., velocity or torque of the motor), which is measured with sensors and processed by the controller. The resulting control signal is used as input to the process, closing the loop.

Closed-loop controllers have the following advantages over open-loop controllers [Kil05]:

- guaranteed performance
- rejection of disturbance (e.g., unmeasured friction in a motor)
- capability to stabilize unstable processes
- reduced sensitivity to parameter variations
- improved performance for reference tracking

A simple controller (see Figure 2.2) attempts to correct the error between a measured process variable (i.e., *output – y(t)*) and a desired setpoint (i.e., *reference – r(t)*) by calculating and then producing a corrective action (i.e., *error – e(t)*) that can adjust the process accordingly.
In particular, the output of the system *y(t)* is fed back to the reference value *r(t)*, through a sensor measurement. The *controller C* then uses the error *e(t)* (i.e., difference between the *reference* and the *output*) to change the inputs *u* to the *system under control P* (e.g., a car).

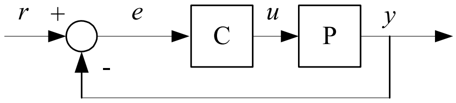


Figure 2.2: A Simple Closed–Loop Controller.

This situation is called a single-input-single-output (SISO) control system. Multi-Input-Multi-Output (MIMO) systems, with more than one input/output, are common. In such cases, variables are represented through vectors instead of simple scalar values. For some distributed parameter systems the vectors may be infinite-dimensional (typically functions).

Common closed-loop controller architecture, widely used in industrial applications, is the proportional-integral-derivative (PID) controller [Kil05]. Its name refers to the three terms operating on the error signal to produce a control signal. Its general form is:

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de}{dt},$$

where: K_P – proportional gain
K_I – integral gain
K_D – derivative gain.

Larger K_P typically means faster response since the larger the error, the larger the proportional term compensation. An excessively large proportional gain will lead to process instability and oscillation. Larger K_I implies steady-state errors are eliminated quicker. The trade-off is larger overshoot. Larger K_D decreases overshoot, but slows down transient response and may lead to instability due to signal noise amplification in the differentiation of the error.

The desired closed-loop dynamics is obtained by adjusting the three parameters K_P, K_I and K_D, usually iteratively by *tuning* and without specific knowledge of a model under control [Kil05].

2.2 Model-based Development

The development process of embedded systems usually occurs on at least three different levels. First a *model* of the system is built. It simulates the required system behavior and usually represents an abstraction of the system. When the model is revealed to be correct, code is generated from the model. This is the software level. Eventually, hardware including the software is the product of the development. The reason for building those intermediate levels is the fact, that it is much cheaper and faster to modify a model than to change the final product. The entire process is called *model-based development* (MBD).
The *multiple V-model* [BN02, SZ06], based on the traditional *V-Modell®*, takes this phenomenon into account. The V-Modell is a guideline for the planning and execution of development projects, which takes into account the whole life cycle of the system. The V-Modell defines the results that have to be prepared in a project and describes the concrete approaches that are used to achieve these results [VM06]. In the *multiple V-model,* each specification level (e.g., model, software, final product) follows a complete V-development cycle, including design, build, and test activities as shown in Figure 2.3. The essence of the *multiple V-model* is that different physical representations of the same system on different abstraction levels are developed, aiming at the same final functionality. Then, the complete functionality can be tested on those different platforms. Since certain detailed technical properties cannot be tested very well on the model, they must be tested on the prototype instead. Testing the various SUT representations often requires specific techniques and a specific test environment. Therefore, a clear relation between the *multiple V-model* and the different test environments exists.

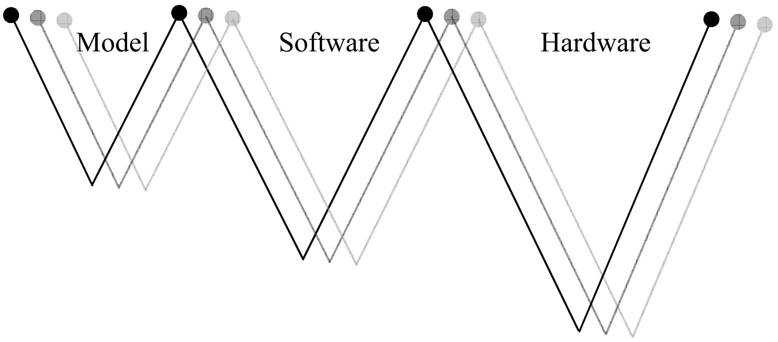


Figure 2.3: The Multiple V-Model.

2.2.1 Issues in Model-based Development

In this thesis the automotive specific *MBD* process is taken into account to illustrate the problems and prepare the background for further considerations on quality assurance (QA).
Since automotive development is an interdisciplinary business including software, electrical and mechanical engineering aspects, it is important to create an executable functional and graphical model. Engineers can find a common functional understanding early in the design phase and improve communication within their teams, with customers, or between car manu-

facturers and suppliers [BN02, Gri03, Leh03, Con04a, Con04b, CFG+05, LK08]. Moreover, separation of concerns appears. The core algorithms are isolated from the technical aspects such as fixed-point scaling (i.e., the transformation of floating-point algorithms to fixed-point computations), calibration data management and the representation of signals in memory [LK08]. That way the complexity of models is still manageable.

This work encompasses the *functional* and *implementation models* (i.e., physical behavioral models) [KHJ07, Con04a] in detail. The objective of a functional model is to demonstrate the feasibility of new functionalities and algorithms. Functional models are executable, but concentrate on the core function of the system. Their testing is reduced to the area of the core functionality. Currently, the tests are nearly automated. Complex functional models are split into sub-models and tested separately. Further on, if it is planned to generate the code automatically from such functional models, they are refined and enhanced to the so-called *implementation models* [KHJ07]. In such a case, they cover all aspects of the final product, except some parts that are usually excluded from being developed using model-based techniques, such as signal I/O code, task scheduling, performance, bootstrap code, or operating system-related functions.
The promise of the model-based design is to simulate, analyze, and validate the models prior to implementation. Executing analysis early in the development cycle enables the detection and correction of design problems sooner at a lower cost.
The most important open issue emerging here is how to assure a high quality of the *functional* and *implementation models*. Any error at the early level is distributed down to the code and hardware realization.

2.2.2 Other Model-based Technologies

The introduction of MBD led to the development of modeling technologies. Consequently executable high-level models can be obtained. The selection of a modeling technology is very dependent on the type of system being modeled and the task for which the model is being constructed [Mos97]. Continuous systems are best modeled by differential equations supplemented by algebraic constraints, if necessary, whereas discrete systems demand Petri nets, finite state automata, Timed Communicating Sequential Processes (Timed CSP) [Mos97].

Some currently available modeling techniques offer a generalized environment and allow for interaction between the methodologies. This enables heterogeneous systems of hybrid nature to be modeled, combined and different views analyzed based on a common notation or environment. The modern techniques provide both support for model structure analysis of a dynamic physical system and a comprehensive, systematic approach to describing differential equations. Moreover, due to their compositional characteristics, they enable hierarchical modeling as well as modifying a particular subsystem to a more detailed model. These technologies can be MATLAB/Simulink/Stateflow (ML/SL/SF), LabView [LabV] and lately also Unified Modeling Language™ (UML®) [UML], or company-specific.
Since graphical modeling languages increase the productivity benefits of 4 to 10 times as given in [KHJ07, Hel+05], several efforts have been undertaken in that area. SCADE Suite™ [DCB04], ASCET [DSW+03], Charon [AGH00], Dymola [Soe00], HYSDEL [TB04], HyVisual [LZ05], Modelica [Mod], hySC [GKS99] are some of the examples.

Additionally, methods such as correct-by-construction (CbyC) methods exist [BFM+05]. They enable automatic code generation ensuring that what is verified on the model level can also be verified on the embedded code level. Applying CbyC testing is still performed, but its role is to validate the correct-by-construction process rather than to find faults. With the growing complexity of software-intense embedded systems, CbyC and formal verification methods become less applicable. It happens because of the increasing state space of the systems. Thus, sample-based testing based on formal correctness criteria and test hypotheses gains importance [BFM$^+$05].

In the next section, ML/SL/SF will be discussed in detail since the approach proposed in this thesis is based on this framework.

2.2.3 MATLAB/Simulink/Stateflow as a Framework

ML/SL/SF[8] is one of the most advanced solutions for modeling embedded systems in the automotive domain. 50% of behavioral models within the control systems are designed applying this tool, being the de-facto standard in the area [Hel$^+$05].

MATLAB: MATLAB (ML) product [MathML] is a technical computing environment, including the *m language*, for analyzing data and developing algorithms. It integrates computation, data analysis, visualization, and programming so as to solve mathematical problems in technical and scientific applications [MathML]. The tool can be used in a number of fields, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, or computational biology.

Simulink: Simulink (SL) product [MathSL] integrated with ML is a software package in the form of a simulation engine and a customizable set of libraries. It is an interactive graphical environment enabling simulation and model-based development of dynamic systems.

SL offers different kinds of solvers for the numerical solution of differential equations and difference equations. SL models are organized as hierarchical block diagrams. Treating an SL model as a *program*, the connections between the blocks would be *variables*, but the value of such a variable would be a *function* defined over a continuum [LN05]. SL models are designed applying blocks available in libraries. *SL libraries* are collections of special-purpose functions that can be re-used in models. With this practice the development time is reduced, function re-usability is supported, and the maintainability is improved. Blocks copied from a library remain linked to their originals such that changes in the originals automatically propagate to the copies in a model. Libraries ensure that the models automatically include the most recent versions of the previously defined blocks and give fast access to the commonly-used functions. Several dedicated libraries for a number of technical and industrial applications exist. A convenient feature of the SL environment is the possibility of its extension by creating new libraries. Another one is the capability of integrating functions written in C using *S-Functions*.

Moreover, if automatic code generation is used to generate C code from the SL/SF model (which is possible by applying e.g., Real-Time Workshop) [Tun04], the functions can be executed in the real vehicle. Such code is run on hardware systems in real time. They are connected

[8] In the context of this thesis the following versions of the software have been used: MATLAB® 7.5.0.342, MATLAB® Report Generator 3.2.1™ (R2007b), Simulink® Report Generator™ 3.2.1 (R2007b), Simulink® 7.0 and Stateflow® 7.0; Release R2007b.

to the real plant by special I/O. As a consequence, changes can be made directly to the function model and tried out by generating code once again [Bod05, BDH05, SCD$^+$07, D-Mint08]. Implementation of the function on a production ECU is also done by automatic or semi-automatic production code generation. However, the requirements on such a code generator are much higher than for *rapid control prototyping* (RCP). The generated code must be highly efficient, error-free, reproducible, and well documented [SCD$^+$07].

A SL model is defined as a tuple $SL = (B, root, sub\_h, P, rlt, sig, subi, subo, C)$:
(i) B is the set of blocks in the model. Subsystem blocks B^s, in-blocks in subsystems B^i, out-blocks in subsystems B^o (representing inputs and outputs of subsystems), merge blocks B^m and blocks with memory B^{mem}. When referring to other types of ”basic” blocks B^b is used in this paper. Furthermore, subsystem can be divided into, normal, virtual subsystems B^{vs} and non-virtual subsystems B^{ns}, $B^s = B^{vs} \cup B^{ns}$. The virtual subsystems do not affect the behavioral semantics of SL, whereas the non-virtual can. Subsystems B^s, in-blocks, B^i and out-blocks B^o are referred to as virtual blocks, since they are used purely for structuring and have no effect on the behavioral semantics;
(ii) $root \in B^{vs}$ is the root subsystem;
(iii) sub_h: $B \rightarrow B^s$ is a function that describes the subsystem hierarchy. For every block b, $sub\_h.b$ gives the subsystem b is in. Note that $sub\_h.root = root$;
(iv) P is the set of ports for inputs and output of data to and from blocks. The ports $P^i \subseteq P$ is the set of in-ports and $P^o \subseteq P$ is the set of out-ports, $P = P^i \cup P^o$;
(v) rlt: $P \rightarrow B$ is a relation that maps every port to the block it belongs to;
(vi) sig: $P^i \rightarrow P^o$ maps every in-port to the out-port it is connected to by a signal;
(vii) $subi$: $B^s \rightarrow P^o \rightarrow \rho(P^i)$ is a partial function that describes the mapping between the in-ports of a subsystem and the out-ports of the non-virtual block B^i representing the in-port block in that subsystem;
(viii) $subo$: $B^s \rightarrow P^o \rightarrow \rho(P^i)$ is a partial function that describes the mapping between the out-ports of a subsystem and the in-ports of the non-virtual block B^o representing the out-port block in that subsystem;
(ix) C is the set of block parameters of the model. The block parameters are a set of constants defined in ML workspace.
Similar definitions of the SL model are given in [BM07, BMW07] including the examples.

Stateflow: Stateflow (SF) product [MathSF] extends the SL so as to support modeling of discrete systems more easily and readably. SF model is sequential and deterministic. It is a hierarchical state machine that includes states labeled with lists of actions and transitions labeled with guards and actions.
The semantic of SF models defined by [Tiw02] is the following. A SF chart is described by a tuple $SF = (D, E, S, T, f)$, where:
(i) $D = D_I \cup D_O \cup D_L$ is a finite set of typed variables that is partitioned into input variables D_I, output variables D_O and local variables D_L;
(ii) $E = E_I \cup E_O \cup E_L$ is a finite set of events that is partitioned into input events E_I, output events E_O and local events E_L;
(iii) S is a finite set of states, where each state is a tuple consisting of three kinds of *actions*: *entry*, *exit*, and *during*; an *action* is either an assignment of an expression to a variable or an *event broadcast*; When a state has parallel (AND) decomposition, all its substates present at the same hierarchy level are always active. When a state has exclusive (OR) decomposition, only one substate can be active at a time.

(iv) 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, *c* $\in$ *WFF*(*D*) is a condition given as a well-formed formula in predicate logic over the variables *D* and *ca, ta* are set of actions (called condition actions and transition actions, respectively);
(v) $f : S \rightarrow (\{and, or\} \times 2^S)$ 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., $s^{root} \notin \cup_i$ *descendants*(s_i), where *descendants*(s_i) is the second component of $f(s_i)$, (b) every non-root 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, $\{s_1, s_2\}$), then the state s is an AND-state consisting of two substates s_1 and s_2. If $f(s)$ = (or, $\{s_1, s_2\}$), then *s* is an OR-state with substates s_1 and s_2. In the syntactic description of an SF chart, junctions are ignored for simplicity.

SL Simulation: The SL execution engine, called a s*olver* is a component that determines the next time step when a simulation needs to meet the target accuracy requirements [MathSL, LN05]. SL provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. They fall into two basic categories fixed-step and variable-step.
Fixed-step solvers solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as *the step size* [MathSL] and will be called *time step size* in the following. Decreasing the time step size increases the accuracy of the results while increasing the time required for simulating the system.
Variable-step solvers vary the time step size during the simulation, reducing the time step size to increase accuracy when a model's states are changing rapidly and increasing the time step size to avoid taking unnecessary steps when the model's states are changing slowly. Computing the time step size adds to the computational overhead at each step but can reduce the total number of steps and hence, simulation time.

When modeling automotive embedded software solvers with a fixed time step size are used [Con04a]. SL provides a set of *fixed-step continuous solvers*. They employ numerical integration to compute the values of a model's continuous states at the current step from the values at the previous step and the values of the state derivatives. This allows the fixed-step continuous solvers to handle models that contain both continuous and discrete states.

In the case studies discussed in this thesis, explicit fixed-step continuous solver *ode4 (i.e., ordinary differential equations of 4th computational complexity)* has been selected. It is based on the integration technique defined by the fourth-order Runge-Kutta (RK4) [PFT+92] formula. This method is reasonably simple and robust. It is a general candidate for numerical solution of differential equations when combined with an intelligent adaptive step-size routine.

Let an initial value problem be specified as follows $y' = f(t, y)$,
where the initial value *y is:* $y(t_0) = y_0$.

Then, the iterative formula for *y* applying the RK4 method is given by the following equations:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{2.1}$$

$$t_{n+1} = t_n + h \tag{2.2}$$

where y_{n+1} is the RK4 approximation of $y(t_{n+1})$ and

$$k_1 = f(t_n, y_n) \tag{2.3}$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \tag{2.4}$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \tag{2.5}$$

$$k_4 = f(t_n + h, y_n + hk_3). \tag{2.6}$$

Thus, the next value (y_{n+1}) is determined by the present value (y_n) plus the product of the size of the interval (h) and an estimated slope.

The slope is a weighted average of slopes:

- k_1 is the slope at the beginning of the interval
- k_2 is the slope at the midpoint of the interval, using slope k_1 to determine the value of y at the point $t_n + h/2$ using Euler's method
- k_3 is again the slope at the midpoint, but now using the slope k_2 to determine the y value
- k_4 is the slope at the end of the interval, with its y value determined using k_3.

In averaging the four slopes, greater weight is given to the slopes at the midpoint:

$$slope = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \tag{2.7}$$

The RK4 method is a fourth-order method, meaning that the error per step is on the order of h^5, while the total accumulated error has order h^4. The above formulas are valid for both scalar- and vector-valued functions (i.e., y can be a vector and f an operator).
Further details about the ML/SL/SF framework can be found in [MathML, MathSL, MathSF].

2.3 Testing

2.3.1 Software Testing

Testing, an analytic means for assessing the quality of software [Wal01, UL06], is one of the most important phases during the software development process with regard to quality assurance. It *„can never show the absence of failures“* [Dij72], but it aims at increasing the confidence that a system meets its specified behavior. Testing is an activity performed for improving the product quality by identifying defects and problems. It cannot be undertaken in isolation. Instead, in order to be in any way successful and efficient, it must be embedded in an adequate software development process and have interfaces to the respective sub-processes.

The fundamental test process according to [BS98, SL05, ISTQB06] comprises (1) planning, (2) specification, (3) execution, (4) recording (i.e., documenting the results), (5) checking for completion, and test closure activities (e.g., rating the final results).
Test planning includes the planning of resources and the laying down of a test strategy: defining the test methods and the coverage criteria to be achieved, the test completion criteria, structuring and prioritizing the tests, and selecting the tool support as well as configuration of the test environment [SL05]. In the test specification the corresponding test cases are specified using the test methods defined by the test plan [SL05]. Test execution means the execution of test cases and test scenarios. Test records serve to make the test execution understandable for people not directly involved (e.g., customer) and prove afterwards, whether and how the planned test strategy was in actual fact executed. Finally, during the test closure step data is collected from completed test activities to consolidate experience, testware, facts, and numbers. The test process is evaluated and a report is provided [ISTQB06].

In addition, [Dai06] considers a process of test development. The test development process, related to steps 2 – 4 of the fundamental test process, can be divided into six phases, which are usually consecutive, but may be iterated: test requirements, test design, test specification, test implementation, test execution, and test evaluation.

The test process aimed at in this work covers with the fundamental one, although only steps 2 – 4 are addressed in further considerations. Compared to [Dai06] the test development process is modified and shortened. It is motivated by the different nature of the considered SUTs. Within traditional software and test development, phases are clearly separated [CH98]. For automotive systems a closer integration of the specification and implementation phases occurs. Hence, after defining the test requirements, the test design phase encompasses the preparation of a test harness. The detailed test specification[9] and test implementation are done within one step as the applied modeling language is executable. Up to this point, the test development process supported in this thesis, is very similar to the one defined by [Leh03]. Further on, test execution and test evaluation are performed simultaneously. The details of the proposed test methodology and test development process will be given in Chapters 4 and 5.

In addition, apart from testing, validation, and verification as further QA activities are especially important in the domain of embedded systems due to the usually high dependability requirements (e.g., safety, reliability, and security). The purpose of validation is to confirm that the developed product meets the user needs and requirements. Verification ensures that it is consistent, complete, and correct at the different steps of the life cycle. Testing means exercising an implementation to detect faults and can be used both for verification and for validation.

A further important aspect is the application of QA for the certification of products, especially in safety-critical domains. New certification standards (e.g., IEC 61508 [IEC05] and ISO 26262 [ISO_SF] for the automotive or the followers of the DO-178B [RT92] in the avionics industry) increasingly require the creation of formal models [Hel$^+$05] and reliable QA techniques.

[9] Test specification phase is called test design in a number of sources [Gri03, ISTQB06, Din08].

2.3.2 Test Dimensions

Tests can be classified in different levels, depending on the characteristics of the SUT and the test system. [Neu04] aims at testing the communication systems and categorizes testing in the dimensions of test goals, test scope, and test distribution. [Dai06] replaces the test distribution by a dimension describing the different test development phases, since she is testing both local and distributed systems. In this thesis embedded systems are regarded as SUTs, thus, the test dimensions are modified as shown in Figure 2.4.

In the following the analysis of the current test process for embedded systems reveals a gap that if bridged, will contribute to the overall development cost and time reduction at most. Herewith, the concrete aims of the test methodology proposed in this thesis, are established.

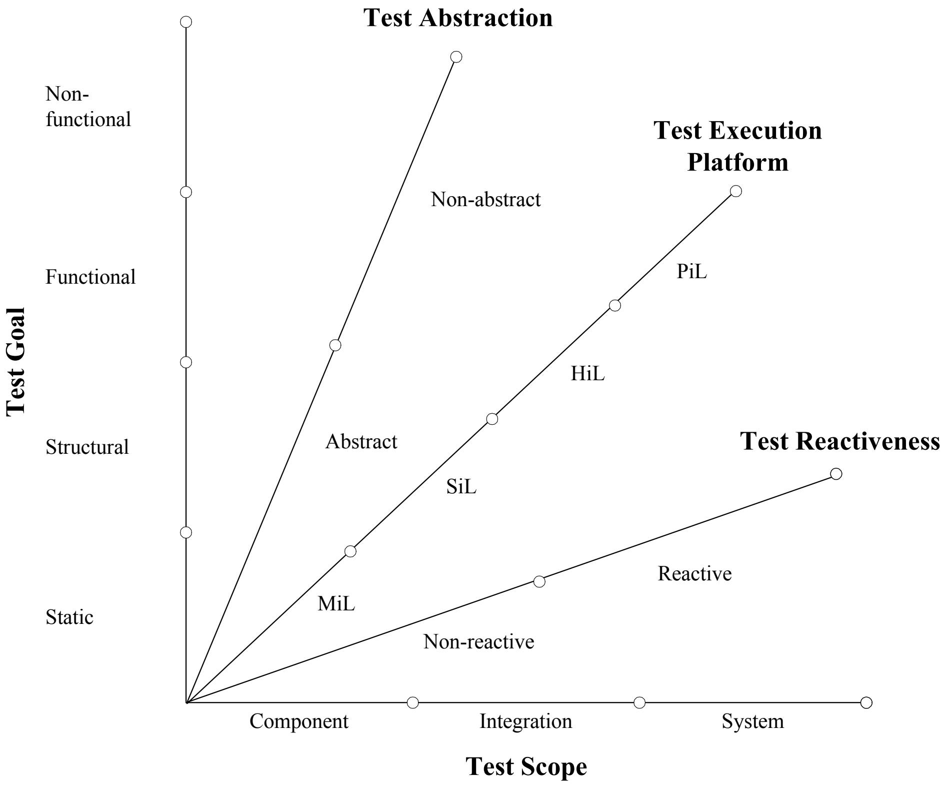


Figure 2.4: The Five Test Dimensions.

Test Goal: During the software development systems are tested with different purposes (i.e., goals). They can be categorized into static testing, also called *review*, and dynamic testing, whereas the latter is distinguished between structural, functional, and non-functional testing. In the automotive, after the *review* phase, the test goal is usually to check the functional behavior of the system. Non-functional tests appear in later development stages.

- *Static Test:* Testing is often defined as the process of finding the errors, failures, and faults. Errors in a program can be revealed without execution by just examining its source code [ISTQB06]. Similarly, other development artefacts can be reviewed (e.g., requirements, models, or test specification itself). This process is called *static testing*. *Dynamic testing* in contrast, bases on execution.

- *Structural Test:* Structural tests cover the structure of the SUT during test execution (e.g., control or data flow). To achieve this, the internal structure of the system (e.g., code or model) needs to be known. Therefore, structural tests are also called white-box or glass-box tests [Mye79, ISTQB06].

- *Functional Test:* Functional testing is concerned with assessing the functional behavior of an SUT against the functional requirements. In contrast to structural tests, functional tests do not require any knowledge about system internals. They are therefore called black-box tests [Bei95]. In this category functional safety tests are also included. Their purpose is to determine the safety of a software product. They require a systematic, planned, executed, and documented procedure. At present, safety tests are only a small part of software testing in the automotive area. By introduction of safety standards such as IEC 61508 [IEC05] and ISO 26262 [ISO_FS] the meaning of software safety tests will, however, increase considerably within the next few years.

- *Non-functional Test*: Similar to functional tests, non-functional tests are performed against requirements specification of the system. In contrast to pure functional testing, non-functional testing aims at the assessment of non-functional, such as reliability, load, or performance requirements. Non-functional tests are usually black-box tests. Nevertheless, for retrieving certain information, e.g., internal clock, internal access during test execution is required.
  For example, during the *robustness test* the system is tested with invalid input data which are outside the permitted ranges to check whether the system is still safe and works properly. As a rule, the robustness is ensured by dedicated plausibility checks integrated into the automotive software.

The focus of this thesis is put on *functional tests*. However some timing[10] and safety aspects are included as well.

[10] In traditional understanding the purpose of *real-time tests* is to find system paths for whose time response of individual tasks or the whole ECU is critical. Since the results of the timing behavior depend strongly on the target architecture, real-time tests are carried out mostly on target systems [Leh03, Con04a, KHJ07].

The context of *real-time testing* in this thesis refers to the situation when the real-time properties are related to functional behavior. In that case they cannot be tested on their own, but require a test case which also involves the associated functional events for stimulating and observing the SUT [Neu04, Dai06]. Thus, real-time testing is incorporated into functional testing and is understood as functional testing of timing constraints rather than real-time properties in the traditional sense.

Test Abstraction: As far as the abstraction level of the test specification is considered, the higher the abstraction, the better test understandability, readability, and reusability is observed. However, the specified test cases must be executable at the same time. The *non-abstract* tests are supported by a number of tool providers (see Chapter 3) and they do not scale for larger industrial projects [LK08]. Hence, the abstraction level should not affect the test execution in a negative way.

This thesis develops a conceptual framework for *abstract* test specification; however, simultaneously an *executable* technical framework for a selected platform is built.

Test Execution Platform: The test execution is managed by so-called test platforms. The purpose of the test platform is to stimulate the test object (i.e., SUT) with inputs, and to observe and analyze the outputs of the SUT.
The test platform is a car with a test driver. The test driver determines the inputs of the SUT by driving scenarios and observes the reaction of the car supported by special diagnosis and measurement hardware/software that records the test data during the test drive and allows the behavior to be analyzed offline. An appropriate test platform has to be chosen depending on the test object, the test purpose, and the necessary test environment.

- *Model-in-the-Loop (MiL):* The first integration level, MiL, is based on the model of the system itself. In this platform the SUT is a functional model or implementation model that is tested in an open-loop (i.e., without any plant model in the first place) or closed-loop test with a plant model (i.e., without any physical hardware) [KHJ07, SZ06, LK08]. The test purpose is basically functional testing in early development phases in simulation environments such as ML/SL/SF.

- *Software-in-the-Loop (SiL):* During SiL the SUT is software tested in a closed or open-loop. The software components under test are usually implemented in C and are either hand-written or generated by code generators based on implementation models. The test purpose in SiL is mainly functional testing [KHJ07]. If the software is built for a fixed-point architecture, the required scaling is already part of the software.

- *Processor-in-the-Loop (PiL):* In PiL embedded controllers are integrated into embedded devices with proprietary hardware (i.e., ECU). Testing on PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor or on a target processor emulator. Tests on PiL level are important because they can reveal faults that are caused by the target compiler or by the processor architecture. It is the last integration level which allows debugging during tests in a cheap and manageable way [LK08]. Therefore, the effort spent by PiL testing is worthwhile in almost all cases.

- *Hardware-in-the-Loop (HiL)*: When testing the embedded system on HiL level the software runs on the final ECU. However the environment around the ECU is still a simulated one. ECU and environment interact via the digital and analog electrical connectors of the ECU. The objective of testing on HiL level is to reveal faults in the low-level services of the ECU and in the I/O services [SZ06]. Additionally, acceptance tests of components delivered by the supplier are executed on the HiL level because the component itself is the integrated ECU [KHJ07]. HiL testing requires real-time behavior of the environment model to ensure that the communication with the ECU is the same as in the real application.

- *Car:* Finally, the last integration level is obviously the car itself, as already mentioned. The final ECU runs in the real car which can either be a sample or a car from the production line. However, these tests, as performed only in late development phases, are expensive and do not allow configuration parameters to be varied arbitrarily [LK08]. Hardware faults are difficult to trigger and the reaction of the SUT is often difficult to observe because internal signals are no longer accessible [KHJ07]. For these reasons, the number of in-car tests decreases while model-based testing gains more attention.

This thesis encompasses mainly the system design level so as to start testing as early as possible in the development cycle. Thus, the *MiL* platform is researched in detail. The other platforms are not excluded from the methodological viewpoint. However, the portability between different execution platforms is beyond the scope of this work.

Test Reactiveness: A concept of *test reactiveness* emerges when test cases are dependent on the system behavior. That is, the execution of a test case depends on what the system under test is doing while being tested. In this sense the system under test and the test driver run in a '*closed loop*'.
In the following, before the *test reactiveness* will be elaborated in detail, the definition of *open-* and *closed-loop system configuration* will be explicitly distinguished:

- *Open-loop System Configuration*: When testing a component in a so-called open-loop the test object is tested directly without any environment or environmental model. This kind of testing is reasonable if the behavior of the test object is described based on the interaction directly at its interfaces (I/O ports). This configuration is applicable for SW modules and implementation sub-models, as well as for control systems with discrete I/O.

- *Closed-loop System Configuration*: For feedback control systems and for complex control systems it is necessary to integrate the SUT with a plant model so as to perform closed-loop tests. In early phases where the interaction between SUT and plant model is implemented in software (i.e., without digital or analog I/O, buses etc.) the plant model does not have to ensure real-time constraints. However, when the HiL systems are considered and the communication between the SUT and the plant model is implemented via data buses, the plant model may include real hardware components (i.e., sensors and actuators). This applies especially when the physics of a system is very crucial for the functionality or when it is too complex to be described in a model.

- *Test Reactiveness*: Reactive tests are tests that apply any signal or data derived from the SUT outputs or test system itself to influence the signals fed into the SUT. With this practice, the execution of reactive test cases varies depending on the SUT behavior. The test reactiveness as such gives the test system a possibility to immediately react to the incoming behavior by modifying the test according to the predefined deterministic criteria. The precondition for achieving the test reactiveness is an online monitoring of the SUT, though. The advantages can be obtained in a number of test specifi-

cation steps (e.g., an automatic sequencing of test cases, online prioritizing of the test cases).
For example, assume that the adaptive cruise control (ACC) activation should be tested. It is possible to start the ACC only when a certain velocity level has been reached. Hence, the precondition for a test case is the increase of the velocity from 0 up to the point when the ACC may be activated. If the test system is able to detect this point automatically, the ACC may be tested immediately.
Although the existing approaches support reactive testing by means of script languages, it is often difficult to understand the test evaluation part of such textually written test cases.
A discussion about possible risks and open questions[11] around reactive tests can be found in [Leh03].

In this work, both *open-* and *closed-loop system configurations* as well as *reactive* and *non-reactive tests* will be regarded. The concept of test reactiveness and the '*closed-loop'* between the SUT and test system will be described in Section 5.6 and instantiated in Section 6.4.

Test Scope: Finally, the test scope has to be considered. Test scopes describe the granularity of the SUT. Due to the composition of the system, tests at different scopes may reveal different failures [ISTQB06, D-Mint08, Wey88]. Therefore, they are usually performed in the following order:

– *Component:* At the scope of component testing, the smallest testable component (e.g., a class in an object-oriented implementation or a single ECU[12]) is tested in isolation.

– *Integration:* The scope of integration test is to combine components with each other and test those not yet as a whole system but as a subsystem (i.e., ACC system composed of a speed controller, a distance controller, switches, and several processing units). It exposes defects in the interfaces and in the interactions between integrated components or systems [ISTQB06].

– *System:* In a system test, the complete system (i.e., a vehicle) consisting of subsystems is tested. A complex embedded system is usually distributed; the single subsystems are connected via buses using different data types and interfaces through which the system can be accessed for testing [Het98].

This thesis encompasses the *component level test*, including both single component and component in-the-loop test, and the *integration level test*.

[11] The main risks are the following: The test might run in a different way than intended by the test designer. In that case, the scenario of interest may not be checked at all. Also, by only slight adjustments of the SUT, the execution flow of a test case may change considerably. A possible solution to these problems would be to introduce monitoring means watching the test execution.

[12] For the purpose of this thesis, the component test scope includes both component testing and component in-the-loop testing. The former applies to a single ECU (e.g., open-loop ECU) testing, the latter holds for a test of an ECU configured together with a plant to form a loop (e.g., closed-loop ECU connected to the car model).

2.3.3 Requirements on Embedded Systems Testing within Automotive

Along with the growing functionality of embedded systems in the automotive and the introduction of model-based development processes, the demands on QA have also increased [LK08]. The QA activities should be systematic, well structured, repeatable, understandable, and possibly automatic. It is a challenge not only because of the time pressure, but also due to the software distribution, its reactive, hybrid nature, and the development process to be obeyed.

Model-based development enables system engineers to test the system in a virtual environment when they are inexpensive. In practice there are just a few testing procedures that address the automotive domain-specific requirements of model-based testing sufficiently [LK08]. As an example, despite the past few years' intensive efforts of automobile manufacturers and their suppliers to enhance the QA of their products, the problems of testing steadily increase in complexity and interconnectedness are still not solved. The variety of proprietary test systems and solutions do not allow an integrated definition, transfer, re-use, and execution of tests for automobile manufacturers, suppliers, and test equipment manufacturers [SG07]. The reason of this state lies in the historical data. About 15 – 20 years ago there was no need for dedicated functional testing methods because the functional complexity was comparatively low and limited mainly to hardware. With the increasing popularity of MBD the engineering discipline of automotive model-based testing has been neglected for a long time [LK08].

The main problems recognized within the existing test solutions encompass the following issues. Only manual test data specification is supported, so that the process of their selection is long and costly. If an automatic generation of test data is possible, then it is based almost only on the criteria resulting from the internal SUT structure. As a consequence, the produced test data are not systematic enough for functional testing. Regarding the test evaluation, entire reference signal flows are needed for the assessment, which are however not available at the early stage of the software development. Only several test patterns exist what implicates the test engineer to start every test project almost from scratch. The entire test specification process is still almost purely manual and no interaction is supported. The issues listed herewith will be discussed in detail based on the concrete test realizations in Sections 3.3 – 3.4.

Hence, new test methodologies should emerge as soon as possible. They should suit the current MBD process and lead to a common understanding of model-based testing concepts. They must enable the testing of the time constraints, discrete and continuous signals as well as their relations. Also, assessment of the expected system behavior should be possible and the reusability of the test specification elements (e.g., in the form of patterns [TYZ05] or libraries) is of high importance.

In particular, the test system should enable the specication and preprocessing of the signals present on the buses including their cycle times in millisecond ranges. The description of signals should proceed on an abstract level. Thus, in this thesis the signal properties (i.e., features constrained with selected predicates) are considered. The timing relations between properties regarding a single signal or a number of signals must be captured. The local and global time concepts are needed. Also, the possibility to define closed-loop tests, called reactive tests [Leh03], is of high importance. The test system should be able to react to events, states, or signal properties in deterministic time (e.g., in real time at hardware level). Watchdog-similar solutions must be applied to diagnose the SUT behavior on the fly. Synchronization of the applied/obtained signals or the test system with the SUT is needed. The specification of the test cases as well as the measurement of events, states, or signal properties need to be time dependent. The test data and their variants should be treated externally. The test evaluation and assessment of the signals should be adjusted to the time concepts, traced, and logged. Even the distribution or a parallel execution of the tests is not out of consideration.

Standard test specification algorithms (e.g., loops for test flow specification, test control for test cases ordering, sleeping or waiting of the test system, test configuration parameterization, or arbitration mechanism) should obviously be supported. Further on, the functional test specification should be split from the concrete SUT implementation by application of test adapters. Besides, an integrated and manufacturer independent test technology is aimed at. Clear relation between the test specification elements and the test objectives should be supported too.

The next important issue is to provide a testing technology which is understandable to a number of stakeholders interested in the development of embedded systems. Hence, a graphical format for the test specification and the implicated model-based testing [BJK$^+$05, UPL06] is demanded. Finally, the quality of the resulting test methodology has to be measured and improved if needed.

The advantages and limitations of the existing test approaches dealing respectively with the set of requirements listed above will be exhaustively discussed in Chapter 3, in particular in Section 3.3.

2.3.4 Definition and Goals of Model-based Testing

Model-based testing (MBT) relates to a process of test generation from an SUT model by application of a number of sophisticated methods. MBT is the automation of black-box test design [UL06]. Several authors [Utt05, KHJ07] define MBT as testing in which test cases are derived in whole or in part from a model that describes some aspects of the SUT based on selected criteria. [Dai06] denotes MBT into model-driven testing (MDT) since she proposes the approach in the context of Model Driven Architecture (MDA). [UPL06] add that MBT inherits the complexity of the domain or, more particularly, of the related domain models.
MBT allows tests to be linked directly to the SUT requirements, makes readability, understandability and maintainability of tests easier. It helps to ensure a repeatable and scientific basis for testing and it may give good coverage of all the behaviors of the SUT [Utt05]. Finally, it is a way to reduce the efforts and cost for testing [PPW$^+$05].

The term *MBT* is widely used today with slightly different meanings. Surveys on different MBT approaches are given in [BJK$^+$05, Utt05, UL06, UPL06, D-Mint08]. In the automotive industry MBT is used to describe all testing activities in the context of MBD [CFS04, LK08]. [Rau02, LBE$^+$04, Con04a, Con04b] define MBT as a test process that encompasses a combination of different test methods which utilize the executable model as a source of information. Thus, the automotive viewpoint on MBT is rather process-oriented. A single testing technique is not enough to provide an expected level of test coverage. Hence, different test methods should be combined to complement each other relating to all the specified test dimensions (e.g., functional and structural testing techniques should be combined). If sufficient test coverage has been achieved on model level, the test cases can be reused for testing the control software generated from the model and the control unit within the framework of back-to-back tests [WCF02]. With this practice, the functional equivalence between executable model, code and ECUs can be verified and validated [CFS04].

For the purpose of this thesis, the following understanding of MBT is used:

Model-based testing is testing in which the *entire test specification* is derived in whole or in part from both *the system requirements and a model* that describe selected functional aspects of the SUT. In this context, the term *entire test specification* covers the *abstract test scenarios* substantiated with the concrete sets of test data and the expected SUT outputs. It is organized in a set of test cases.

Further on, the resulting test specification is *executed* together with the SUT model so as to provide the test results. In [Con04a, CFS04] no additional models are being created for test purposes, but the already existent functional system models are utilized for the test. In the test approach proposed in this thesis (see Chapters 4 – 5) the system models are exploited too. In addition, however, a *test specification model* (also called *test case specification*, *test model,* or *test design* in the literature [Pre03b, ZDS$^+$05, Dai06]) is created semi-automatically. Concrete test data variants are derived automatically from it.

Moreover, since the MBT approaches have to be integrated into the existing development processes and combined with the existing methods and tools, in this thesis *ML/SL/SF* has been selected as both system and test modeling framework and execution platform. By that, MBD and MBT are supported using the same environment.

2.3.5 Patterns

The concept of patterns emerges when software reusability is targeted. A pattern is a general solution to a specific recurring design problem. It explains the insight and good practices that have evolved to solve a problem. It provides a concise definition of common elements, context, and essential requirements for a solution as discussed in [Ale79, VS04, Neu04, PTD05]. Patterns were first used to describe constellations of structural elements in buildings and towns [Ale79].
The purpose of software patterns is to capture software design know-how and make it reusable. They can enhance the structure of software and simplify its maintenance. Patterns also improve communication among software developers and empower less experienced engineers to produce high-quality results. They contribute the efficiency due to a common and understandable vocabulary for problem solutions that they provide [Neu04].

Similarly to the manner software patterns contribute to the software development process, *test patterns* enhance progress in testing. *Test patterns* [VS04] represent a form of reuse in test development in which the essences of solutions and experiences gathered in testing are extracted and documented so as to enable their application in similar contexts that might arise in the future. Test patterns aim at capturing test design knowledge from past projects in a canonical form, so that future projects would benefit from it.

2.4 Summary

The first aim of this chapter was to discuss the backgrounds of embedded systems and their development. Herein, the details on their different aspects have been presented. Further on, electronic control units and the fundamentals on control theory have been provided. They are recalled in this thesis afterwards, especially in Chapters 4 – 6, so as to contribute to the proposed QA framework and complete the case studies.

Besides this, MBD concepts applied in the automotive domain have been introduced. Basic knowledge on the ML/SL/SF framework has been given, including the algorithm of its simulation solver. These details are needed throughout this work, starting from Chapter 4, in order to define the test specification in SL/SF language. Then, several MBD approaches have also been reviewed.

Furthermore, testing concerns have been outlined. Herewith, testing – categorized by the dimensions of test scopes, test goals, test abstraction, test execution platform, test configuration, and test reactiveness – has revealed the complexity of the considered domain. The emphasis of this thesis is put on functional, abstract, executable, and reactive MiL level tests. The details on their specification are given mainly in Chapters 4 – 5. The test execution is exemplified in Chapter 6.

Finally, the concept of patterns has been provided so as to introduce an abstract way of the test specification in the upcoming chapters.

3 Selected Test Approaches

"Contradiction is not a sign of falsity,
nor the lack of contradiction a sign of truth."

- Blaise Pascal

This chapter reviews related work on the model-based testing (MBT) of embedded, hybrid real-time systems. Firstly, in Section 3.1, an overview of *taxonomy* for MBT, introduced initially by [UPL06], is analyzed and extended for the needs of the considered domain. Further on, constraints on nature of the embedded system models are explicitly given. Afterwards, the particular *categories* of the *taxonomy* are discussed in detail. These relate to test generation, test execution, and test evaluation.

In Section 3.2, a short report on the current test trends recognized in the automotive domain is outlined. Then, in Section 3.3, a *trapezoid* narrowing the range of MBT approaches is formed. It is derived based on the analysis of the *test dimensions* given in Section 2.3.1 and the *test categories* of the MBT *taxonomy*. Several test approaches are selected for further analysis. Finally, a comparison of the selected solutions is provided and their challenges and limitations are pointed out.

A comprehensive list of the corresponding test tools available in the academia or industry is presented in Appendix A. Moreover, a brief description of the test method proposed in this thesis is given. Finally, conclusions are taken. A summary completes the chapter.

3.1 Categories of Model-based Testing

In [UPL06, UL06] a comprehensive *taxonomy* for MBT identifying its three general *classes*: model, test generation, and test execution is provided. Each of the classes is divided into further *categories*. The model-related ones are subject, independence, characteristics, and paradigm. Further on, the test generation is split into test selection criteria and technology, whereas the test execution partitions into execution options.

In the following work, the *taxonomy* is enriched with an additional *class*, called test evaluation. The test evaluation means comparing the actual system under test (SUT) outputs with the expected SUT behavior based on a test oracle. Test oracle enables a decision to be made as to whether the actual SUT outputs are correct. It is, apart from the data, a crucial part of a test case. The test evaluation is divided into two *categories*: specification and technology.

Furthermore, in this thesis only one selected class of the system model is investigated. For clarification purposes, its short description based on the options available in the taxonomy of [UPL06, UL06] will be given. The subject is the model (e.g., Simulink/Stateflow (SL/SF) model) that specifies the intended behavior of the SUT and its environment, often connected via a feedback loop. Regarding the independence level this model can be generally used for both test case[13] and code generation. Indicating the model characteristics, it provides deterministic hybrid behavior constrained by timed events, including continuous functions and various data types. Finally, the modeling paradigm combines a history-based, functional data flow paradigm (e.g., SL function blocks) with a transition-based notation (e.g., SF charts).

The overview of the resulting, slightly modified and extended MBT *taxonomy* is illustrated in Figure 3.1. The modification results from the focus of this thesis, which is put on embedded systems. All the categories are split into further instances which influence each other within a given category or between them. The notion of 'A/B/C' at the leaves indicates mutually exclusive options, while the straight lines link further instantiations of a given dimension without exclusion. It is a good practice since, for example, applying more than one test selection criterion and by that, more generation technologies can provide a better test coverage, eventually.

[13] In the test approach proposed in this thesis (see Chapter 4), firstly, a test specification model is created semi-automatically and then the test data variants forming the test cases are derived automatically out of the test model.

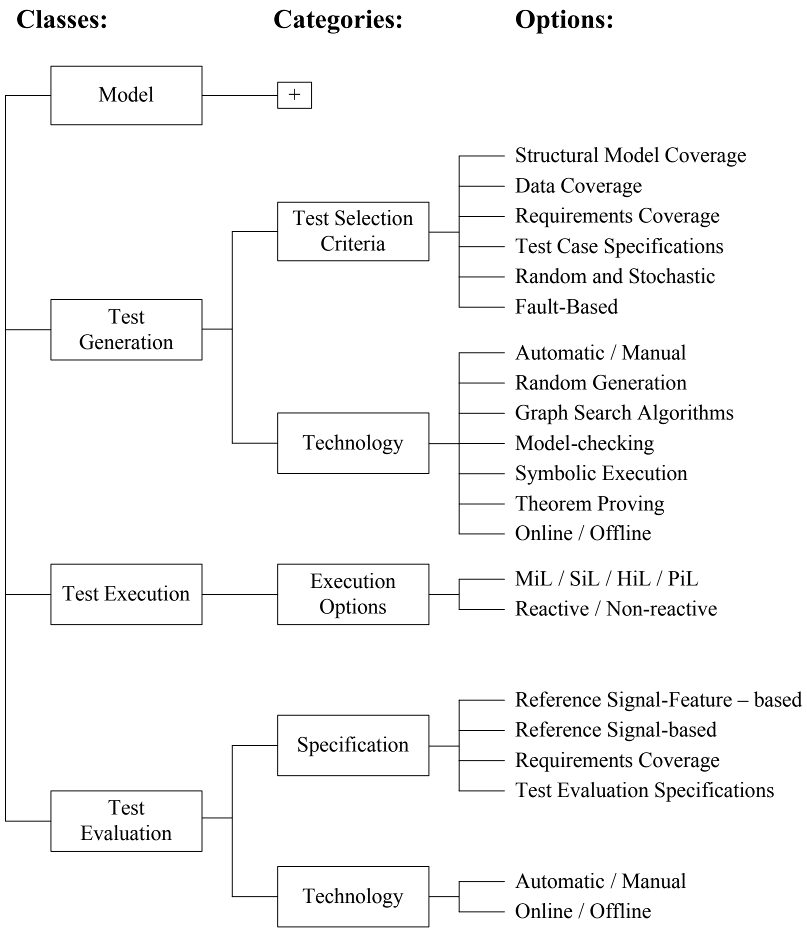


Figure 3.1: Overview of the Taxonomy for Model-based Testing.

In the next three sections the *classes* of the MBT taxonomy are referred to and the particular *categories* and *options* are explained in depth. The descriptions of the most important *options* following in this thesis contain examples of their realization, respectively.

3.1.1 Test Generation

The process of test generation starts from the system requirements, taking into account the test objectives. It is defined in a given test context and leads to the creation of test cases. A number of approaches exist depending on the test selection criteria and generation technology. They are reviewed below.

Test selection criteria: Test selection criteria define the facilities that are used to control the generation of tests. They help to specify the tests and do not depend on the SUT code [UL06]. In the following, the most commonly-used criteria are investigated. Referring to the discussion given in Section 2.4.4, different test methods should be combined to complement each other so as to achieve the best test coverage. Hence, there is no best suitable criterion for generating the test specification.

- *Structural model coverage criteria:* These exploit the structure of the model to select the test cases. They deal with coverage of the control-flow through the model, based on ideas from control-flow through code.
  In [Pre03] it is shown how test cases can be generated that satisfy the Modified Condition/Decision Coverage (MC/DC) coverage criterion. The idea is to first generate a set of test case specifications that enforce certain variable valuations and then generate test cases for them.
  Similarly, Safety Test Builder (STB) [STB] or Reactis Tester [ReactT, SD07] generate test sequences covering a set of SF test objectives (e.g., transitions, states, junctions, actions, MC/DC coverage) and a set of SL test objectives (e.g., boolean flow, look-up tables, conditional subsystems coverage) (see Section 3.3 for more detail).

- *Data coverage criteria:* The idea is to split the data range into equivalence classes and select one representative from each class. This partitioning is usually complemented by the boundary value analysis [KLP$^+$04], where the critical limits of the data ranges or boundaries determined by constraints are additionally selected.
  An example is the MATLAB Automated Testing Tool (MATT) [MATT] enabling black-box testing of SL models and code generated from it by Real-Time Workshop®. It generally enables the creation of custom test data for model simulations by setting their types for each input. Further on, accuracy, constant, minimum, and maximum values can be provided to generate the test data matrix.
  Another realization of this criterion is provided by Classification Tree Editor for Embedded Systems (CTE/ES) [CTE] implementing the Classification Tree Method (CTM) [GG93, Con04a]. The SUT inputs form the classifications in the roots of the tree. Then, the input ranges are divided into classes according to the equivalence partitioning method. The test cases are specified by selecting leaves of the tree in the combination table. A line in the table specifies a test case. CTE/ES provides a way of finding test cases systematically. It breaks the test scenario design process down into steps. Additionally, the test scenario is visualized in a graphical user interface (GUI).

- *Requirements coverage criteria:* These aim to cover all the informal SUT requirements. Traceability of the SUT requirements to the system or test model/code can support the realization of this criterion. It is targeted by almost every test approach.

- *Test case specifications:* When the test engineer defines a test case specification in some formal notation, these can be used to determine which tests will be generated. It is explicitly decided which set of test objectives should be covered. The notation used to express these objectives may be the same as the notation used for the model [UPL06]. Notations commonly used for test objectives include FSMs, UML Testing Profile (UTP) [UTP], regular expressions, temporal logic formulas, constraints, and Markov chains (for expressing intended usage patterns).
  A prominent example of applying this criterion is described in [Dai06], where the test case specifications are retrieved from UML® models and transformed into executable tests in Testing and Test Control Notation, version 3 (TTCN-3) [ETSI07] by using

Model Driven Architecture (MDA) [MDA] methods [ZDS$^+$05]. The work of [Pre03, Pre04] is also based on this criterion (see symbolic execution in the next paragraph).

- *Random and stochastic criteria:* These are mostly applicable to environment models, because it is the environment that determines the usage patterns of the SUT. A typical approach is to use a Markov chain to specify the expected SUT usage profile. Another example is to use a statistical usage model in addition to the behavioral model of the SUT [CLP08]. The statistical model acts as the selection criterion and chooses the paths, while the behavioral model is used to generate the oracle for those paths.
  Exemplifying, Markov Test Logic (MaTeLo) [MaTL] can generate test suites according to several algorithms. Each of them optimizes the test effort according to the objectives such as boundary values, functional coverage, and reliability level. Test cases are generated in XML/HTML format for manual execution or in TTCN-3 for automatic execution [DF03].
  Another instance, Java Usage Model Builder Library (JUMBL) [JUMB] can generate test cases either as a collection of test cases which cover the model with the minimum cost or by random sampling with replacement, or in order by probability, or by interleaving the events of other test cases. There is also an interactive test case editor for creating test cases by hand.

- *Fault-based criteria:* These rely on knowledge of typically occurring faults, often designed in the form of a fault model.

Test generation technology: One of the most appealing characteristics of model-based testing is its potential for automation. The automated generation of test cases usually necessitates the existence of kind of test case specifications [UPL06].

- *Automatic/Manual technology:* Automatic test generation refers to the situation when the test cases are generated automatically from the information source based on the given criteria. Manual test generation refers to the situation when the test cases are produced by hand.

- *Random generation:* Random generation of tests is done by sampling the input space of a system. It is easy to implement, but it takes a long time to reach a certain satisfying level of model coverage as [Gut99] reports.

- *Graph search algorithms:* Dedicated graph search algorithms include node or arc coverage algorithms such as the Chinese Postman algorithm[14], which covers each arc at least once. For transition-based models, which use explicit graphs containing nodes and arcs, there are many graph coverage criteria that can be used to control test generation. The commonly used are all nodes, all transitions, all transition pairs, and all cycles. The method is exemplified by [LY94], additionally based on structural coverage of FSM models.

[14] Chinese Postman algorithm, http://www.uclic.ucl.ac.uk/harold/cpp/ [04/20/08].

- *Model checking:* Model checking is a technology for verifying or falsifying properties of a system. A property typically expresses an unwanted situation. The model checker verifies whether this situation is reachable or not. It can yield counter examples when a property is not satisfied. If no counter example is found, then the property is proven and the situation has never been reached. Such a mechanism is implemented in CheckMate [ChM, SRK$^+$00], Safety Checker Blockset (SCB) [SCB], or in Embedded-*Validator* [EmbV].

  The general idea of test case generation with model checkers is to first formulate test case specifications as reachability properties, for instance, "eventually, a certain state is reached or a certain transition fires". A model checker then yields traces that reach the given state or that eventually make the transition fire. Other variants use mutations of models or properties to generate test suites.

- *Symbolic execution:* The idea of symbolic execution is to run an executable model not with single input values but with sets of input values instead [MA00]. These are represented as constraints. With this practice, symbolic traces are generated. By instantiation of these traces with concrete values the test cases are derived. Symbolic execution is guided by test case specifications. These are given as explicit constraints and symbolic execution may be done randomly by respecting these constraints.

  In [Pre03b] an approach to test case generation with symbolic execution on the backgrounds of Constraint Logic Programming (CLP), initially transformed from the AutoFocus models [AuFo], is provided. [Pre03b, Pre04] concludes that test case generation for both functional and structural test case specifications limits to finding states in the model's state space. Then, the aim of symbolic execution of a model is then to find a trace representing a test case that leads to the specified state.

- *Theorem proving:* Usually theorem provers are used to check the satisfiability of formulas that directly occur in the models. One variant is similar to the use of model checkers where a theorem prover replaces the model checker.

  The technique applied in Simulink® Design Verifier™ (SL DV) [SLDV] uses mathematical procedures to search through the possible execution paths of the model so as to find test cases and counter examples.

- *Online/Offline generation technology:* With online test generation, algorithms can react to the actual outputs of the SUT during the test execution. This idea is used for implementing the reactive tests too.

  Offline testing means that test cases are generated before they are run. A set of test cases is generated once and can be executed many times. Also, the test generation and test execution can be performed on different machines, levels of abstractions, or in different environments. Finally, if the test generation process is slower than test execution, then there are obvious advantages to doing the test generation phase only once.

3.1.2 Test Execution

The test execution options in the context of this thesis have been already described in Section 2.4.2. Hence, in the following only reactive testing and the related work on the reactive/non-reactive option is reviewed.

Execution options: Execution options refer to the execution of a test.

- *Reactive/Non-reactive execution:* Reactive tests are tests that apply any signal or data derived from the SUT outputs or test system itself to influence the signals fed into the SUT. Then the execution of reactive test cases varies depending on the SUT behavior, in contrast to the non-reactive test execution, where the SUT does not influence the test at all.
  Reactive tests can be implemented within AutomationDesk [AutD]. Such tests react to changes in model variables within one simulation step. The scripts run on the processor of the HiL system in real time, synchronously to the model.
  The Reactive Test Bench [WTB] allows for specification of single timing diagram test benches that react to the user's Hardware Description Language (HDL) design files. Markers are placed in the timing diagram so that the SUT activity is recognized. Markers can also be used to call user-written HDL functions and tasks within a diagram.
  [DS02] conclude that a dynamic test generator and checker are more effective in creating reactive test sequences. They are also more efficient because errors can be detected as they happen. Resigning from the reactive testing methods, a simulation may run for a few hours only to find out during the post-process checking that an error occurred a few minutes after the simulation start.
  In [JJR05], in addition to checking the conformance of the implementation under test (IUT), the goal of the test case is to guide the parallel execution towards satisfaction of a test purpose. Due to that feature, the test execution can be seen as a game between two programs: the test case and the IUT. The test case wins if it succeeds in realizing one of the scenarios specified by the test purpose; the IUT wins if the execution cannot realize any test objective. The game may be played offline or online [JJR05].

3.1.3 Test Evaluation

The test evaluation, also called the test assessment, is the process that exploits the test oracle. It is a mechanism for analyzing the SUT output and deciding about the test result. As already discussed before, the actual SUT results are compared with the expected ones and a verdict is assigned. An oracle may be the existing system, test specification, or an individual’s specialized knowledge. The test evaluation is treated explicitly in this thesis since herewith a new concept for the test evaluation is proposed.

Specification: Specification of the test assessment algorithms may be based on different foundations that cover some criteria. It usually forms a kind of model or a set of ordered reference signals/data assigned to specific scenarios. Considering continuous signals the division into reference-based and reference signal-feature – based evaluation becomes particularly important:

- *Reference signal-based specification:* Test evaluation based on reference signals assesses the SUT behavior comparing the SUT outcomes with the previously specified references.
  An example of such an evaluation approach is realized in the MTest [MTest, Con04a] or SystemTest™ [STest]. The reference signals can be defined using a signal editor or they can be obtained as a result of a simulation. Similarly, test results of back-to-back tests can be analyzed with the help of MEval [MEval, WCF02].

- *Reference signal-feature – based specification:* Test evaluation based on reference signal feature[15] assesses the SUT behavior comparing the SUT outcomes partitioned into features with the previously specified reference values for those features.
  Such an approach to test evaluation is supported in the Time Partitioning Test (TPT) [TPT, Leh03, LKK^{+}06]. It is based on the script language Python extended with some syntactic test evaluation functions. By that, the test assessment can be flexibly designed and allows for dedicated complex algorithms and filters to be applied to the recorded test signals. A library containing complex evaluation functions is available.

- *Requirements coverage criteria:* Similar to the case of test data generation, they aim to cover all the informal SUT requirements, but this time with respect to the expected SUT behavior (i.e., regarding the test evaluation scenarios) specified there. Traceability of the SUT requirements to the test model/code can support the realization of this criterion.

- *Test evaluation specifications:* This criterion refers to the specification of the outputs expected from the SUT after the test case execution. Already authors of [ROT98] describe several approaches to specification-based test selection and build them up on the concept of test oracle, faults and failures. When the test engineer defines test scenarios in some formal notation, these can be used to determine how, when and which tests will be evaluated.

Technology: The technology of the test assessment specification enable an automatic or manual process, whereas the execution of the test evaluation occurs online or offline.

- *Automatic/Manual technology:* The *option* can be understood twofold, either from the perspective of the test evaluation definition, or its execution. Regarding the specification of the test evaluation, when the expected SUT outputs are defined by hand, then it is a manual test specification process. In contrast, when they are derived automatically (e.g., from the behavioral model), then the test evaluation based on the test oracle occurs automatically. Usually, the expected reference signals/data are defined manually; however, they may be facilitated by parameterized test patterns application.
  The activity of test assessment itself can be done manually or automatically.
  Manual specification of the test evaluation means is supported in Simulink® Verification and Validation™ (SL VV) [SLVV], where the predefined assertion blocks can be assigned to the test signals defined in the Signal Builder block in SL. With this practice, functional requirements can be verified during model simulation. The evaluation itself then occurs automatically.

[15] A signal feature (called also signal property in [GW07, SG07, GSW08]) is a formal description of certain defined attributes of a signal. In other words, it is an identifiable, descriptive property of a signal. It can be used to describe particular shapes of individual signals by providing means to address abstract characteristics of a signal. Giving some examples: increase, step response characteristics, step, maximum etc. are considerable signal features [ZSM06, GW07, SG07, GSW08, ZXS08]. A feature can be predicated by other features. Generally, predicates on signals (or on signal features), temporal fragmentation of the signal or temporal relation between more than one signal (or signal feature) are distinguished. This definition will be extended and clarified in Section 4.1.

The tests developed in SystemTest exercise MATLAB (ML) algorithms and SL models. The tool includes predefined test elements to build and maintain standard test routines. Test cases, including test assessment, can be specified manually at a low abstraction level. A set of automatic evaluation means exists and the comparison of obtained signals with the reference ones is done automatically.

– *Online/Offline execution of the test evaluation:* The online (i.e., on the fly) test evaluation happens already during the SUT execution. Online test evaluation enables the concept of test control and test reactiveness to be extended. Offline means the opposite. Hence, the test evaluation happens after the SUT execution.
Watchdogs defined in [CH98] enable online test evaluation. It is also possible when using TTCN-3. TPT [Leh03] means for online test assessment are limited and are used as watchdogs for extracting any necessary information for making test cases reactive. The offline evaluation is more sophisticated in TPT.

Tools realizing the selected test approaches can be classified according to the criteria listed in the MBT taxonomy. A comprehensive list of the MBT tools from academia and industry is also provided in Appendix A. In Section 3.3 a short description of the automotive trends is reported and in Section 3.4 the comparison of test approaches is analyzed.

3.2 Automotive Practice and Trends

Established test tools from, e.g., dSPACE GmbH [dSP], Vector Informatik GmbH [VecI], MBTech Group [MBG] etc. are highly specialized for the automotive domain and usually come together with a test scripting approach which is directly integrated to the respective test device. All these test definitions pertain to a particular test device and by that not portable to other platforms and not exchangeable.

Recently, the application of model-based specifications in development enables more effective and automated process reaching a higher level of abstraction.
Thereby, model-based testing and platform-independent approaches have been developed such as CTE/ES [Con04a], MTest, and TPT [Leh03]. As already mentioned CTE/ES supports the CTM with partition tests according to structural or data-oriented differences of the system to be tested. It also enables the definition of sequences of test steps in combination with the signal flows and their changes along the test. Because of its ease of use, graphical presentation of the test structure, and the ability to generate all possible combination of tests, it is widely used in the automotive domain. Integrated with the MTest, test execution, test evaluation, and test management become possible. After the execution, SUT output signals can be compared with previously obtained reference signals. MTest has, however, only limited means to express test behaviors which go beyond simple sequences, but are typical for control systems. The test evaluation bases only on the reference signals which are often not yet available at the early development phase yet and the process of test development is fully manual.
TPT addresses some of these problems. It uses an automaton-based approach to model the test behavior and associates with the states pre- and post-conditions on the properties of the tested system (including the continuous signals) and on the timing. In addition, a dedicated run-time environment enables the execution of the tests. The test evaluation is based on a more sophisticated concept of signal feature. However, the specification of the evaluation happens in Python

language, without any graphical support. TPT is a dedicated test technology for embedded systems controlled by and acting on continuous signals, but the entire test process is manual and difficult to learn.

Current research work aims at designing a new platform-independent test specification language, one of the branches called TTCN-3 continuous[16] [SBG06, BKL07, SG07, GSW08]. Its fundamental idea is to obtain a domain-specific test language, which is executable and unifies tests of communicating, software-based systems in all of the automotive subdomains (e.g., telematics, power train, body electronics, etc.) integrating the test infrastructure as well as the definition, and documentation of tests. It should keep the whole development and test process efficient and manageable. It must address the subjects of test exchange, autonomy of infrastructure, methods and platforms, and the reuse of tests.

TTCN-3 has the potential to serve as a testing middleware. It provides concepts of local and distributed testing. A test solution based on this language can be adapted to concrete testing environments. However, while the testing of discrete controls is well understood and available in TTCN-3, concepts for specification-based testing of continuous controls and for the relation between discrete and the continuous system parts are still under ongoing research [SBG06, SG07, GSW08].

This option becomes interesting, especially in the context of a new paradigm – AUTomotive Open System Architecture (AUTOSAR)[17] that has been observed as an automotive development trend for the last few years. Traditional TTCN-3 is already in use to test discrete interactions within this architecture. The remaining hybrid or continuous behavior could be tested with TTCN-3 embedded.

Another graphical test specification language being already in the development stage is UML Testing Profile for Embedded Systems [DM_D07, D-Mint08, Liu08]. Its backgrounds root from UTP, TPT, and Model-in-the-Loop for Embedded System Test, abbreviated as MiLEST (the approach proposed in this thesis). These are coordinated and synchronized with the concepts of TTCN-3 embedded too.

Apart from the tools commonly known in the automotive industry, further approaches exist and are applied for testing the embedded systems. Referring to the constraints[18] on the topic of this thesis, a further comprehensive analysis is done in the upcoming section.

3.3 Analysis and Comparison of the Selected Test Approaches

In the following, numerous test approaches are analyzed. Firstly, several, randomly selected academic achievements on testing embedded systems are considered, in general. Then, the test

[16] The resulting profile will be called TTCN-3 embedded, http://www.temea.org/ [04/22/08].

[17] AUTOSAR is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. AUTOSAR Consortium, http://www.autosar.org [04/22/08].

[18] The emphasis of this thesis is put on functional, abstract, executable and reactive MiL level tests of hybrid embedded systems (see also Section 2.4).

methods restricted by a concrete set of criteria (cf. Figure 3.2) and applied in the industry are compared.

3.3.1 Analysis of the Academic Achievements

The approach, of which the realization is called Testing-UPPAAL [MLN03], presents a framework, a set of algorithms, and a tool for the testing of real-time systems based on symbolic techniques used in the UPPAAL model checker. The timed automata network model is extended to a test specification. This one is used to generate test primitives and to check the correctness of system responses. Then, the retrieved timed traces are applied so as to derive a test verdict. Here, online manipulation of test data is an advantage and this concept is partially reused in MiLEST (cf. test reactiveness on the test data level in Section 5.5.1). After all, the state-space explosion problem experienced by many offline test generation tools is reduced since only a limited part of the state space needs to be stored at any point in time. The algorithms use symbolic techniques derived from model checking to efficiently represent and operate on infinite state sets. The implementation of the concept shows that the performance of the computation mechanisms is fast enough for many realistic real-time systems [MLN03]. However, the approach does not deal with the hybrid nature of the system at all.

Similar as in MiLEST the authors of [BKB05] consider that a given test case must address a specific goal, which is related to a specific requirement. The proposed approach computes one test case for one specific requirement. This strategy avoids handling the whole specification at once, which reduces the computation complexity. However, here again, the authors focus on testing the timing constraints only, leaving the hybrid behavior testing open.

The authors of [CLP08] use two distinct, but complementary, concepts of sequence-based specification (SBS) and statistical testing. The system model and the test model for test case generation are distinguished, similar as in MiLEST. The system model is the black-box specification of the software system resulting from the SBS process. The test model is the usage model that models the environment producing stimuli for the software system as a result of a stochastic process. The framework proposed in this approach automatically creates Markov chain test models from specifications of the control model (i.e., SF design). The test cases with an oracle are built and statistical results are analyzed. Here, the formerly mentioned JUMBL methods are applied [Pro03]. Statistics are used as a means for planning the tests and isolating errors with propagating characteristics. The main shortcoming of this work is that mainly SF models are analyzed, leaving the considerable part of continuous behavior open (i.e., realized in SL design). This is not sufficient for testing the entire functionality of the systems considered in this thesis.

In contrast, the authors of [PHPS03] present an approach to generating test cases for hybrid systems automatically. These test cases can be used both for validating models and verifying the respective systems. This method seems to be promising, although as a source of test information two types of system models are used: a hybrid one and its abstracted version in the form of a discrete one. This practice may be very difficult when dealing with the continuous behavior described purely in SL.

The authors of [PPW^{+}05] evaluate the efficiency of different MBT techniques. They apply the automotive network controller case study to assess different test suites in terms of error detection, model coverage, and implementation coverage. Here, the comparison between manually or automatically generated test suites both with and without models, at random or with dedicated functional test selection criteria is aimed at. As a result, the test suites retrieved from models, both automatically and manually, detect significantly more requirements errors than handcrafted test suites derived only from the requirements. The number of detected programming errors does not depend on the use of models. Automatically generated tests find as many errors as those defined manually. A sixfold increase in the number of model-based tests leads to an 11% increase [PPW$^+$05] in detected errors.

Algorithmic testbench generation (ATG) technology [Ole07], though commercially available, is an interesting approach since here the test specification is based on the rule sets. These rule sets show that the high-level testing activities can be performed as a series of lower-level actions. By that, an abstraction level is introduced. This hierarchical concept is also used in MiLEST while designing the test system. ATG supports some aspects of test reactiveness, similar to MiLEST, and includes metrics for measuring the quality of the generated testbench specification. Finally, it reveals cost and time reduction while increasing the quality of the SUT as claimed in [Ole07].

3.3.2 Comparison of the Test Approaches Applied in the Industry

Based on the analysis of the *test dimensions* given in Section 2.4.1 and the *test categories* of the MBT taxonomy (see Section 3.2) a *trapezoid* shown in Figure 3.2 is derived. It narrows the range of the test approaches that are compared in the further part of this section in detail.

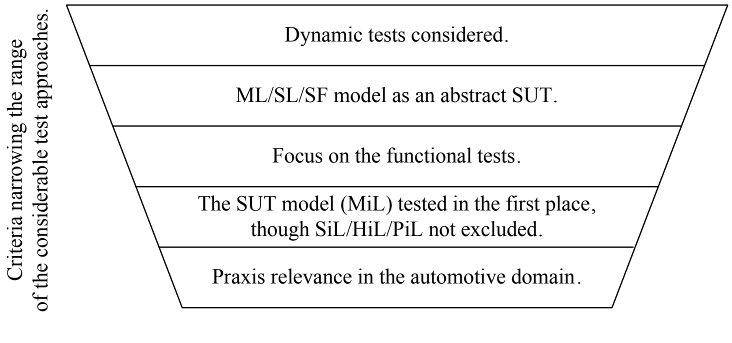

Criteria narrowing the range of the considerable test approaches.
Dynamic tests considered.
ML/SL/SF model as an abstract SUT.
Focus on the functional tests.
The SUT model (MiL) tested in the first place, though SiL/HiL/PiL not excluded.
Praxis relevance in the automotive domain.

Figure 3.2: Trapezoid Selecting the Range of the Test Approaches.

Firstly, in this thesis only dynamic tests are considered, leaving the static reviews open. Several researches on static testing are to be found in [AKR$^+$06, ALS$^+$07, SDG$^+$07, FD07]. Further on, only the *class* of models defined in Section 3.2 is investigated. ML/SL/SF is selected as the environment enabling instantiation of such models. Although structural test methods have been briefly outlined, the focus of this thesis is put on functional testing. Finally, having the previous constraints in mind, praxis relevant approaches are to be elaborated.
Hence, the test approaches influencing this work conceptually and technically are analyzed in Table 3.1 and Table 3.2 in terms of the MBT taxonomy. The characteristic of the approach encompassed in this thesis is introduced at the end of this chapter.

Table 3.1: Classification of the Selected Test Approaches based on the MBT Taxonomy.

MBT Categories, Options	*Test Generation*		*Test Execution*	*Test Evaluation*	
Selected Test Tools	*Test Selection Criteria*	*Technology*	*Execution Options*	*Specification*	*Technology*
Embedded*Validator* [EmbV]	- does not apply[19]	- automatic generation - model checking	- MiL, SiL - non-reactive	- requirements coverage	- manual specification - does not apply
MEval [MEval]	- does not apply since here back-to-back regression tests are considered	- does not apply	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
MTest with CTE/ES [MTest, CTE]	- data coverage - requirements coverage - test case specification - offline generation	- manual generation	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
Reactis Tester [ReactT]	- structural model coverage - offline generation	- automatic generation - model checking[20]	- MiL, SiL, HiL - non-reactive	- test evaluation specifications	- automatic specification - offline evaluation
Reactis Validator [ReactV]	- structural model coverage - requirements coverage - offline generation	- automatic generation - model checking	- MiL, SiL - non-reactive	- test evaluation specifications	- manual specification - online evaluation
Simulink® Verification and Validation™ [SLVV]	- does not apply	- manual generation	- MiL - non-reactive	- requirements coverage	- manual specification - online evaluation
Simulink® Design Verifier™ [SLDV]	- structural model coverage - offline generation	- automatic generation - theorem proving	- MiL, SiL - non-reactive	- requirements coverage - test evaluation specifications	- manual specification - online evaluation

[19] Unless otherwise noted, the expression *'does not apply'* is used when the particular option is not explicitly named for the given test approach. In that case either further deep investigation is needed to assess the option or the assessment plays no role for further analysis.

[20] The tool employs an approach called *guided simulation* to generate quality input data automatically. The idea behind this approach is to use algorithms and heuristics so as to automatically obtain inputs covering the targets (i.e., model elements to be executed at least once). The author decided to classify this approach as a sophisticated variant of model checking technology.

SystemTest™ [STest]	- data coverage - offline generation	- automatic generation[21]	- MiL, SiL, HiL - non-reactive	- reference signals-based	- manual specification - offline evaluation
TPT [TPT]	- data coverage - requirements coverage - test case specification - offline and online generation	- manual generation	- MiL, SiL, PiL, HiL - reactive	- reference signal-feature – based	- manual specification - online and offline evaluation
T-VEC [TVec]	- structural model coverage - data coverage - requirements specification - offline generation	- automatic generation	- MiL, SiL - non-reactive	- test evaluation specifications [ROT98]	- automatic specification - does not apply

Considering the test approaches introduced in Table 3.1, several diversities may be observed. Embedded*Validator* [EmbV, BBS04] uses model checking as test generation technology and thus, is limited to discrete model sectors. The actual evaluation method offers a basic set of constraints for extracting discrete properties, not addressing continuous signals. Only a few temporal constraints are checked. However, the mentioned properties of the model deal with the concept on signal features, whereas the basic verification patterns contribute to the test patterns and their reusability within the technique proposed in this thesis.

MEval [MEval] is especially powerful for back-to-back-tests and for regression tests, since even very complex reference signals are already available in this case. The option is excluded from further consideration.

MTest [MTest] with its CTE/ES [CTE], as already mentioned in the previous section, gives a good background for partitioning of test inputs into equivalence classes. The data coverage and test case specifications criteria are reused in MiLEST to some extent. Similarly as in SystemTest, the test evaluation is based only on reference signal-based specification, which constitutes a low abstraction level, thus it is not adopted for further development.

Reactis Tester [ReactT], T-VEC [TVec], or the method of [Pre03] present approaches for computing test sequences based on structural model coverage. It is searched for tests that satisfy MC/DC criteria. Their value is that the test suites are generated for units (functions, transitions) but also for the entire system or on integration level. Although the methods seem to be very promising due to their scope and automation grade, this thesis is focused on functional testing only. Structural testing is left open as a complementary method.

[21] The test vectors may be defined using MATLAB expressions or generated randomly applying probability distributions for Monte Carlo simulation [DFG01].

In Reactis Validator [ReactV, SD07] only two predefined validation patterns are available. Hence, a systematic test specification is not possible. This gap is bridged in MiLEST that provides *assertion – precondition* pairs. They enable the test evaluation functions to be related with the test data generation.
For SL DV [SLDV] a similar argumentation applies, although another test generation technology is used. An advantage of these three solutions is their possibility to cover both functional and structural test goals, at least to some extent.
SL VV [SLVV] gives the possibility of implementing a test specification directly next to the actual test object, but the standard evaluation functions cover only a very limited functionality range, a test management application is missing and test data must be created fully manually. A similar test assessment method, called 'watchdog' and 'observer', has been introduced by [CH98, DCB04], respectively.

TPT [TPT], as discussed in the previous section, is platform-independent and can be used at several embedded software development stages, which is not directly supported with MiLEST, although extensions are possible. It is the only tool from the list in Table 3.1 that enables reactive testing and signal-feature – based specification of the test evaluation algorithms. These concepts are reused and extended in the solution proposed in this thesis.

The classification of the selected test approaches based on the test dimensions can be derived from the discussion above. However, the summary is given explicitly for complementary purposes. Further details can be found in Appendix A annexed to this thesis.

Table 3.2: Classification of the Selected Test Approaches based on the Test Dimensions.

Test Dimensions *Selected Test Tools*	*Test Goal*	*Test Abstraction*	*Test Execution Platform*	*Test Reactiveness*	*Test Scope*
Embedded*Validator*	- functional	- abstract	- MiL, SiL[22]	- non-reactive	- component - integration
MEval	- functional	- non-abstract	- MiL, SiL, PiL, HiL	- non-reactive	
MTest with CTE/ES	- functional	- semi-abstract	- MiL, SiL, PiL, HiL	- non-reactive	
Reactis Tester	- structural	- non-abstract	- MiL, SiL, HiL	- non-reactive	
Reactis Validator	- functional	- abstract	- MiL, SiL	- non-reactive	
Simulink® Verification and Validation™	- functional	- abstract	- MiL	- non-reactive	
Simulink® Design Verifier™	- structural - functional	- abstract	- MiL, SiL	- non-reactive	
SystemTest™	- functional	- non-abstract	- MiL, SiL, HiL	- non-reactive	
TPT	- functional	- abstract	- MiL, SiL, PiL, HiL	- reactive	
T-VEC	- structural	- non-abstract	- MiL, SiL	- non-reactive	

[22] For SiL, PiL and HiL test adapters and test drivers are usually needed.

The main *shortcomings* and *problems* within the existing test solutions are the following:

- Automatic generation of test data is based almost only on structural test criteria or state-based models (e.g., SF charts), thus it is not systematic enough.
- For functional testing only manual test data specification is supported, which makes the test development process long and costly.
- The test evaluation is based mainly on the comparison of the SUT outputs with the entire reference signal flows. This demands a so-called *golden device* to produce such references and makes the test evaluation not flexible enough.
- Only a few test patterns exist. They are not structured and not categorized.
- The entire test development process is still almost only manual.
- Abstraction level is very low while developing the test design or selecting the test data variants.

Finally, none of the reviewed test approaches overcomes all the shortcomings given above at once. Based on the recognized problems and the criteria that have been proven to be advantageous in the reviewed related work, the first shape of MiLEST may be outlined. MiLEST deals with all the listed problems. In particular, the following are in focus:

- Systematic and automatic test data generation process is supported. Here, not only a considerable reduction of manual efforts is advantageous, but also a systematic selection of test data for testing functional requirements including such system characteristics as hybrid, time-constrained behavior is achieved. By that, the method is cheaper and more comprehensive than the existing ones.
- The test evaluation is done based on the concept of signal feature, overcoming the problem of missing reference signals. These are not demanded for the test assessment any more.
- A catalog of classified and categorized test patterns is provided, which eases the application of the methodology and structures the knowledge on the test system being built.
- Some of the steps within the test development process are fully automated, which represents an improvement in the context of the efforts put on testing.
- A test framework enabling the specification of a hierarchical test system on different abstraction levels is provided. This gives the possibility to navigate through the test system easily and understand its contents immediately from several viewpoints.

A brief description of the MiLEST method is given below, whereas a report on its main contributions in relation to the related work is given in Table 3.3 and will be discussed in Chapters 4 – 6 in depth.

The application of the same modeling language for both system and test design brings positive effects. It ensures that the method is relatively clear and it does not force the engineers to learn a completely new language. Thus, MiLEST is a SL add-on exploiting all the advantages of SL/SF application. It is a test specification framework, including reusable test patterns, generic graphical validation functions (VFs), test data generators, test control algorithms, and an arbitration mechanism collected in a dedicated library. Additionally, transformation functions in the form of ML scripts are available so as to automate the test specification process. For running the tests, no additional tool is necessary. The test method handles continuous and discrete signals as well as timing constraints.

Table 3.3: Classification of the Test Approaches based on the Selected Criteria.

Criteria *Selected Test Methodologies, Technologies, Tools*	*Test Specification*				*Test Patterns Support*	*Transformation and Automation Facilities*
	Manual Test Case / Test Data Specification	***Automatic Test Case / Test Data Generation***	***Test Evaluation Scenarios as Driving Force***	***Formal Verification***		
Embedded*Validator*				+	+ (15 patterns)	
MTest with CTE/ES	+					
Reactis Tester		+		+		
Reactis Validator		+	+		–/+ (2 patterns)	
Simulink® Verification and Validation™	+		+		+ (12 patterns)	
Simulink® Design Verifier™		+		+	–/+ (4 patterns)	
SystemTest™	+					
TPT	+				+	
T-VEC		+		+		
Transformations Approach [Dai06]	+					+
Watchdogs [CH98]			+			
MiLEST		+	+		+ (over 50 patterns)	+

A starting point applying the method is to design the test specification model in MiLEST. Further on, generic test data patterns are retrieved automatically from some marked portions of the test specification. The test data generator concretizes the data. Its functionality has some similarities to the CTM method and aims at systematic signal production. The SUT input partitions and boundaries are used to find the meaningful representatives. Additionally, the SUT outputs are considered too. Hence, instead of searching for a scenario that fulfills the test objective it is assumed that this has already been achieved by defining the test specification. Further on, the method enables to deploy a searching strategy for finding different variants of such scenarios and a time point when they should start/stop.

Since at the early stage of new system functionalities development reference signals are not available, another solution has to be provided. In this thesis a new method for describing the SUT behavior is given. It is based on the assessment of particular signal features specified in the requirements. For that purpose a novel, abstract understanding of a *signal* is defined. This is the fundamental contribution of this work as both test case generation and test evaluation are based on this concept. Numerous signal features are identified; feature extractors, comparators, and feature generators are implemented. Due to their application, the test evaluation may be performed online which enables an active test control, opens some perspectives for test generation algorithms and provides extensions of reactive testing, but at the same time reduces the performance of the test system. Also, new ways for first diagnosis activities and failure management are possible.

Finally, the introduced reactive testing concept relates to the test control, but it is more powerful, especially in the context of hybrid systems. [Leh03] defines the test reactiveness as a reaction of the test data generation algorithm on the SUT outputs during the test execution. In particular, the test case reacts to a defined SUT state, instead of on a defined time point. This definition is extended in this thesis as the test data can be additionally influenced by signals from the test evaluation. Combining this method with the traditional test control definition, the sequence of test cases execution can be organized and test data generation can be influenced depending on the verdict of the previous test case (as in TTCN-3); depending on the SUT outputs (as in TPT and in TTCN-3) and on other test evaluation signals (e.g., reset, trigger, activation).

The *options* that MiLEST covers with respect to the MBT taxonomy are listed in Table 3.4.

Table 3.4: Classification of MiLEST with respect to the MBT Taxonomy.

Test Approach	*Test Generation: Selection Criteria and Technology*	*Test Execution Options*	*Test Evaluation: Specification and Technology*
MiLEST	- data coverage - requirements coverage - test case specifications - automatic generation - offline generation	- MiL - reactive	- reference signal-feature – based - requirements coverage - test evaluation specifications - automatic and manual (depending on the process step) - online evaluation

In this chapter, the first set of questions arisen in the introduction to this thesis has been considered. The role of *system model* in relation to the quality-assurance process has been established. Since black-box testing is aimed at the availability of the system model and an access to its interfaces became the crucial issue. It has been decided to provide a *test model* including all the parts of a complete test specification. Also, a *common language* for both system and test specifications have been used.

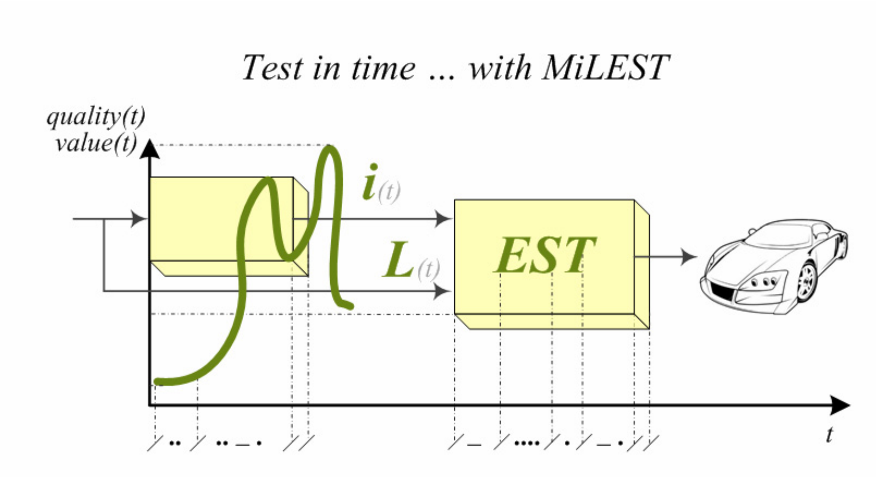
3.4 Summary

In this chapter related work on MBT has been introduced and its analysis has been done. At first, MBT *taxonomy* has been elaborated, extended and presented on a diagram. Then, the system model has been fixed as a concrete instantiation of one of the *categories* from the *taxonomy*. Further *categories* and *options* from the *taxonomy* have been discussed in detail. They are related to the test generation, test execution and test evaluation.
Then, in Section 3.2, the current testing situation identified in the automotive domain has been reported. In Section 3.3, a *trapezoid* has been introduced so as to narrow the range of MBT approaches investigated further on. Then, an analysis of the selected test methods followed, resulting in a list of the corresponding test tools available in the academia or industry. Their comprehensive classification of the MBT solutions has been attached in Appendix A as a table. Hence, it should be referred to while analyzing the contents of this chapter. For additional work on MBT the reader is linked to the surveys given in [BJK$^+$05, UL06].

Finally, based on the analysis of challenges and limitations of the existing approaches a short characteristic of MiLEST has been elaborated, which will be followed in Chapters 4 – 6 in depth.

The work related to the subject of this thesis will be recalled in the upcoming chapters many times. This practice should serve as an explanation of fundamental concepts of MiLEST. In particular, related research on test patterns will be given Chapter 4 due to its strong relation to the approach proposed in this thesis. The same applies to the considerations on signal features.

– Part II –



4 A New Paradigm for Testing Embedded Systems

"Anybody who has been seriously engaged in scientific work of any kind realizes that over the entrance to the gates of the temple of science are written the words: "You must have faith." It is a quality which the scientist cannot dispense with."

- Max Karl Ernst Ludwig Planck

While adding new functionalities into the existing systems in the automotive domain, model-based development (MBD) paradigm is often applied. At the early stage of this sort of development, a functional system model, usually executable, serves as a means for introducing the novelties. Neither real-world nor reference signals are available for testing yet. This implies a need for another solution.

In this thesis, a methodology for *model-based testing* of the embedded, hybrid, real-time system functionality is provided. It is based on the current software development trends from the practice. In the *scope of this methodology*, a new manner for the stimulation and evaluation of the system behavior is given. It is breaking the requirements down into characteristics of specific *signal features*. Based on them, the test specification and test assessment are realized. For that purpose, a novel understanding of a *signal* is defined that allows for its abstract description. This enables to have a new view on automatic test signals generation and evaluation, both of them being the fundamental contributions of this thesis.
As an outcome, the manual efforts for test data generation are reduced and, at the same time, their systematic selection is achieved. This makes the proposed method cheaper and more comprehensive than the existing approaches.
Then, because of the application of *signal features,* the test evaluation is overcoming the problem of missing reference signals. These are not demanded for the test assessment any more, which shifts the current common practice towards a new testing paradigm.
Many of the steps within the test development process are fully automated, which gives a significant improvement in the context of the efforts put on testing, especially comparing to the other, still very manual methods.

In the following chapter the conceptual contents of the method proposed in this thesis are introduced. Its realization is called Model-in-the-Loop for Embedded System Test and abbreviated as MiLEST. In Section 4.1, the basics on signal and signal feature are presented. Also, different combinations of such features are investigated. Then, in Section 4.2, they are classified according to the availability in time and assessment delay. Every entry of this classification is reviewed from the perspective of signal generation and signal evaluation. At any one time, a realization proposal is given. This gives the possibility to deal with an automatic test generation and evaluation and by that, the manual efforts spent on such activities are considerably reduced. Section 4.3 introduces the concept of test patterns that are applicable in the proposed method and outlines the characteristics of the selected solution. The collection of the MiLEST test patterns is attached as a table in Appendix B. These ease the application of the methodology and structure the knowledge on the test system being built. As a consequence, flexible test specifications can be obtained systematically. In Section 4.4, process of test specification and test execution is briefly described. Finally, Section 4.5 provides related work on the test paradigms applied in this thesis, whereas Section 4.6 completes the chapter with a summary.

4.1 A Concept of Signal Feature

Before the signal-feature approach and features classification will be presented, fundamental knowledge on signal and signal processing is given. Additionally, also logical connectives and temporal relations between features are introduced for completeness of the discussion on test specification.

4.1.1 A Signal

A signal is any time-varying or spatial-varying quantity [KH96, NM07]. It represents a pattern of variation of some form [EMC⁺99]. Signals are variables that carry information.
Mathematically, signals are represented as a function of one or more independent variables. As a matter of example, a black and white video signal intensity is dependent on x, y coordinates and time t, which is described as $f(x,y,t)$.

In this thesis, exclusively signals being a function of time – $f(t)$ will be concerned. Giving some examples: velocity of a car over time is classified as a mechanical signal, voltage, and current in a circuit are electrical signals and acoustic pressure over time belongs to acoustic signals [KH96].

Signals can be categorized in various ways. The distinction regarded in this thesis is reduced to the difference between discrete and continuous time that the functions are defined over and between their discrete and continuous values. Most signals in the real world are time-continuous, as the scale is infinitesimally fine. Differential equations are used for representing how continuous signals are transformed within time-continuous systems, whereas difference equations enable discrete signals to be transformed within most time-discrete systems. An analog signal is a kind of signal that is continuously variable in time, as opposed to digital signal which is varying in a limited number of individual steps along its range [KH96]. Analog signal differs from a digital signal in that small fluctuations in the signal are meaningful. Digital signals are value-discrete, but are often derived from an underlying value-continuous physical process.

Furthermore, in this thesis causal systems are discussed. A system is causal if the output at a time only depends on input values up to that time, not on the future values. It is referred to as non-anticipative, because the system output does not anticipate future values of the input [KH96, EMC$^+$99].

Mathematically, a system $x(t) \rightarrow y(t)$ is causal if $x_1(t) \rightarrow y_1(t),\ x_2(t) \rightarrow y_2(t)$ and given two input signals $x_1(t), x_2(t)$ have the relation $x_1(t) = x_2(t)$ for all $t \leq t_0$, then $y_1(t) = y_2(t)$ for all $t \leq t_{0.}$
All real-time physical systems are causal, because time only moves forward, i.e., effect occurs after cause [EMC$^+$99].

In Simulink (SL) even the events are described using signals with a value at every *simulation time step*. The realization of time-continuous and time-discrete signals is based on the same principle. Time-discrete signals are a subset of analog signals, although their representation seems to be redundant from some viewpoints.

Moreover, a continuous signal in SL is actually *time-sampled continuous signal* represented as an equation:

$x[k] = x(kT)$, where:

- T is sample time[23]
- k is the *simulation time step*.

Hence, the prototypical realization of the test technology introduced in this thesis limits its scope to the generation, controlling, and assessment of *value-continuous* and *time-sampled continuous signals*, although the system behavior expressed by those signals can be of continuous and discrete nature. Thus, theoretically, the separation of concerns still applies and the signals will be considered as both *time-continuous* and *time-discrete*.

To sum up, the test system proposed hereby utilizes the basic principles of signal processing[24]. Subsequently, the two main tasks, namely test data generation and test evaluation, can be renamed to *signal generation* and *signal evaluation* with respect to the lowest level of abstraction as presented in Figure 4.1. The system specification is used as a decisive factor about the type of signals that are generated and evaluated within functional tests. Additionally, the feedback path supports the test reactiveness (cf. Section 3.4).

[23] The *sample time* also known as *simulation time step size* is called *step size* in ML/SL/SF [MathSL]. In this thesis it will be denominated to *time step size*.

[24] Signal processing is the analysis, interpretation and manipulation of signals.

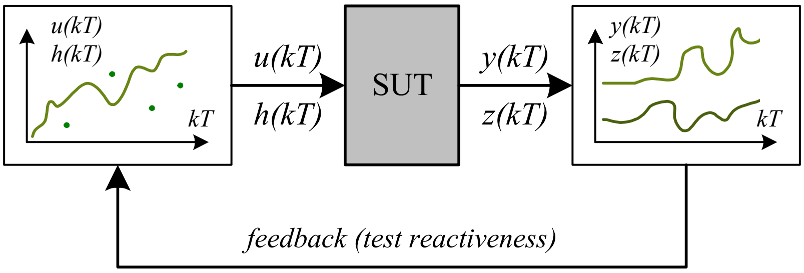


Figure 4.1: A Hybrid Embedded System with Discrete- and Continuous-timed Signals.

4.1.2 A Signal Feature

A signal feature (SigF), also called signal property in [GW07, SG07, GSW08], is a formal description of certain predefined attributes of a signal. In other words, it is an identifiable, descriptive property of a signal. It can be used to describe particular shapes of individual signals by providing means to address abstract characteristics of a signal. Giving some examples: *step response characteristics*, *step*, *minimum* etc. are considerable SigFs.
Whereas the concept of SigF is known from the signal processing theory [KH96, Por96], its application in the context of software testing has been revealed by [Leh03, LKK$^+$06, ZSM06, MP07, GW07, SG07, GSW08, ZXS08]. In this thesis, the SigF is additionally considered as a means for test data generation and, similar to [Leh03], evaluation of the SUT outputs.

Graphical instances of SigFs are given in Figure 4.2. The signal presented on the diagram is fragmented in time according to its descriptive properties resulting in: *decrease*, *constant*, *increase*, *local maximum*, *decrease*, and *step response*, respectively. This forms the backgrounds of the solution presented in this work.

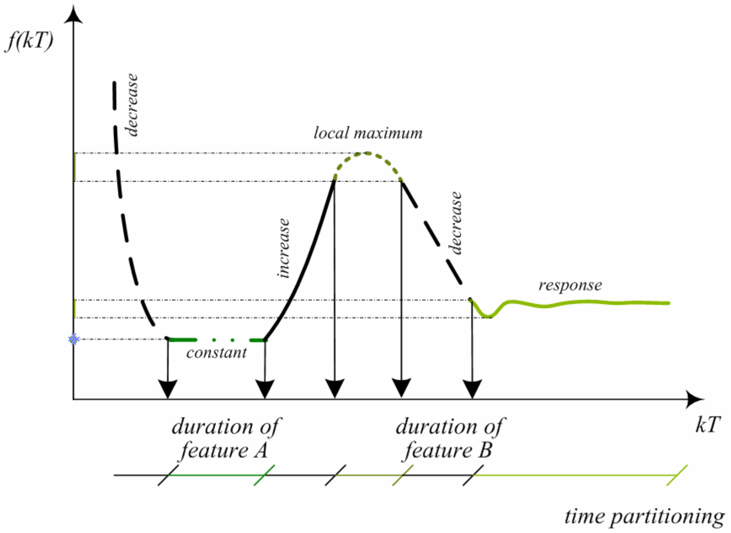


Figure 4.2: A Descriptive Approach to Signal Feature.

A feature can be predicated by other features, logical connectives, or timing relations.

In mathematics, a predicate is either a relation or the boolean-valued function that amounts to the characteristic function or the indicator function of such a relation [Cop79, Men97]. A function *P: X→ {true, false}* is called a predicate on X or a property of X. In this sense, a SigF is a predicate on a signal. Sometimes it is inconvenient or impossible to describe a set by listing all of its elements. Another useful way to define a set is by specifying a property that the elements of the set have in common. *P(x)* notation is used to denote a sentence or statement P concerning the variable object *x*. The set defined by *P(x)* written {*x* | *P(x)*}, is a collection of all the objects for which P is sensible and true. Hence, an element of {*x* | *P(x)*} is an object *y* for which the statement *P(y)* is true.

For instance, assuming that the object is a *SigF A* over the signal sig_A and the predicate *P(SigF A)* is a maximum, the following is obtained: {*SigF A*(sig_A) | *SigF A*(sig_A) is a maximum} is the set of all maximums within the signal sig_A, called here *setMax*(sig_A).

Further on, looking for a particular value of the maximum: {*setMax*(sig_A) | *setMax*(sig_A) is equal to *v*} outputs a set of such maximums that are equal to value *v*.

4.1.3 Logical Connectives in Relation to Features

A logical connective[25] is a logical constant which represents a syntactic operation on well-formed formulas. It can be seen as a function which maps the truth-values of the sentences to which it is applied [Cop79, CC00].

The basic logical connectives applied in the prototypical realization of the test method in this thesis are the following, also listed in [Cop79]:

- negation (NOT) – (¬)
- conjunction (AND) – (&)[26]
- disjunction (OR) – (||)
- material implication (IF … , THEN …) – (→)

They are used for formalization of the test specification. The detailed description of their application follows in Section 4.4 and in Section 5.2.

4.1.4 Temporal Expressions between Features

A temporal relation is an inter-propositional relation that communicates the simultaneity or ordering in time of events, states, or SigFs. In logic, the term temporal logic is used to describe

[25] A logical connective is also called a truth-functional connective, logical operator or propositional operator.

[26] For boolean arguments, the single ampersand ("&") constitutes the (unconditional) "logical AND" operator, called logical conjunction [Men97], while the double ampersand ("&&") is the "conditional logical AND" operator. That is to say that the single ampersand always evaluates both arguments whereas the double ampersand will only evaluate the second argument if the first argument is true. – as retrieved from: http://www.jguru.com/faq/view.jsp?EID=16530 [04/24/2008].

any system of rules and symbolism for representing and reasoning about propositions qualified in terms of time.

In this thesis a set of temporal expressions from these available in [BBS04, GW07, GHS$^+$07] has been chosen for introducing the relations between features:

- *after* (d_1, d_2) B – *SigF B* occurs after time specified in the range between d_1 and d_2, *where* $0 <= d_1 <= d_2$
- *after* (A) B – if *SigF A* occurs, *SigF B* occurs afterwards
- *before* (A)B – if *SigF A* occurs, than *SigF B* must have occurred before
- *during* (d_1, d_2) B – *SigF B* occurs continuously during time specified in the range between d_1 and d_2
- *during* (A)B – if *SigF A* occurs, *SigF B* occurs continuously during activation of *SigF A*
- *every* nth occurrence (A) – *SigF A* occurs every n^{th} time
- *within* (d_1, d_2) B – *SigF B* occurs at least once in time specified in the range between d_1 and d_2
- *within* (A)B – if *SigF A* occurs, *SigF B* occurs at least once whereas *SigF A* is active

The selection has been done based on the experience analysis from model checking as described in [GW07]. The test specification may be combined from SigFs, logical operators, and temporal expressions. Giving an example: *IF* A *& after(*d_1*,* d_2*)* B || C*, THEN during(*d_3*,* d_4*)* D – means that if *SigF A* on sig_A holds and after a period of time not lower than d_1 and not higher than d_2 *SigF B* on sig_B holds or *SigF C* on sig_C holds, then *SigF D* on sig_D should hold during the time period starting from d_3 until d_4.

Additionally, the following selected temporal expressions [KM08] are covered by the corresponding statements:

- *always(*A*)* is covered by *IF* true *THEN* A
- *never(*A*)* is covered by *IF* true *THEN* ¬A.
- *eventually(*A*)* is covered by *IF* true *THEN* A at least once[27]

Similar to the logical connectives, the temporal relations are used for formalization of the test specification. The description of their application follows in Section 4.4 and in Section 5.2.

The combinations of predicates can be defined either between features characterizing one signal or more signals, using both logical connectives and temporal relations. For the examples below, the following assumption is valid: A_1 and A_2 are SigFs characterizing a time-continuous signal sig_A. A and B are SigFs characterizing two different time-continuous signals, *C* represents a SigF characterizing a discrete signal sig_C.

[27] At least once means in this context at least once until the end of the predefined simulation time for a particular model.

- *within*(A$_1$)B$_1$&A$_2$ – if *SigF A$_1$* occurs, *SigF B$_1$* and *SigF A$_2$* occur together at least once whereas *SigF A$_1$* is active
- *after*(y ms)A&B – *SigF A* and *SigF B* occur together after *y* milliseconds
- *during*(A)B&C – if *SigF A* occurs, *SigF B* and *SigF C* occur both continuously during activation of *SigF A*
- A||$\neg C$ – *SigF A* or no *SigF C* occurs
- A=v – a set of *SigFs A* (e.g., maximum) which values are equal to *v*.

4.2 Signal Generation and Evaluation

4.2.1 Features Classification

This thesis is driven by the practicability factor. Thus, the classification of SigF is done following the realization algorithms, instead of any other theoretical approach.
This trend is motivated by the facts that MiLEST is based on the already existing modeling platform and its implications contribute to the reasoning about features. Moreover, a principal idea of this work is to show the feasibility of the proposed solution relating to the running case studies. Thus, the implementation behind the conceptual reasoning is in the primary focus.

The fundamental task of signal processing, in the context of the approach proposed in this work, is to include the concept of SigF. Hence, the core problems of *signal generation* and *signal evaluation* are limited to the generation of an appropriate SigF or a combination of SigFs over a predefined signal, on the one hand; and evaluation of an extracted SigF from a signal, on the other hand. Therefore, the activities of performing those practices are sometimes denominated as *feature generation* and *feature extraction*, respectively. *Feature extraction* is a mechanism for reducing the information about signal evaluation. This enables the test assessment be abstracted from the large sequences of values that signals represent. *Feature extraction* is also called *feature detection* in this thesis.

Generation of a feature characterizing a signal translates to the generation of a specific signal, which contains the particular properties. The concept of generating the signals relates to the mechanisms which serve for their extraction, and thus evaluation. The features extraction perspective is used for their classification. In fact, SigFs could be categorized applying the generation viewpoint too, but it is of more value to use the other perspective. This kind of practice simplifies the process of understanding the entire test specification. Moreover, it is motivated by the fact, that the specification and evaluation part, including feature extraction, must be designed by an engineer, whereas the signal generation part is done fully automatically based on the test specification model. Thus, the starting point is to get familiar with the mechanisms of feature extraction, in fact. An additional classification would cause only an abstract overhead for the end-user.

Nevertheless, before SigFs will be categorized in detail, a brief overview on the scheme of feature generation will be given. Firstly, a default signal shape is defined for every SigF (cf. Figure 4.3). Then, the range of permitted values for the signal is defined. Further on, a minimal duration time of the feature is provided, in case needed. Otherwise, a default duration time is set. Finally, feature specifics are introduced in terms of the so-called *generation information*. For example, a step generation includes a size of the step as shown in Figure 4.3, whereas an increase generation includes the shape of the increase, a slope, initial and final values. Additional *parameters* that need to be taken into account while feature generation relate to the evaluation mechanism for a particular feature. They must be set following the values of the same parameters that have been applied in the extraction part. A simple example is a step, for which the duration of constant signal appearing before the step, must be set. Otherwise, the feature detection mechanism could not work properly. Then, generating the step, the duration of the generated constant signal, must be set on the minimal value specified within the extraction so as to be detectable at all.

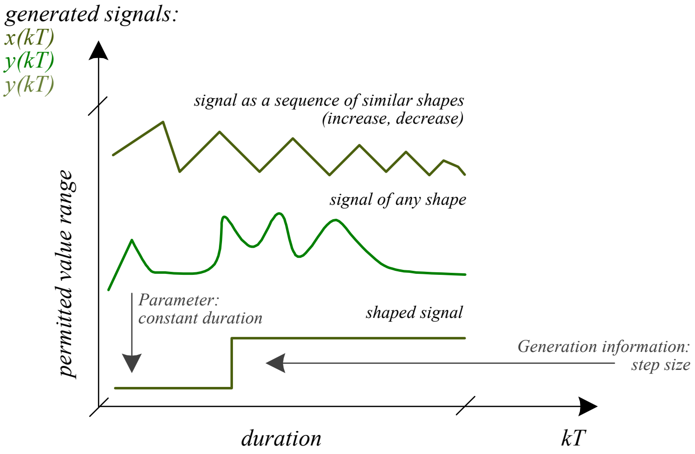


Figure 4.3: Signal-Features Generation – a few instances.

To sum up, a generic pattern for *signal generation* is always the same – a feature is generated over a selected signal and the parameters are adjusted according to a predefined algorithm (cf. Figure 4.4); however, some feature specifics must be included for an actual generation of every single SigF. The details concerning the abstract considerations on that subject and their realization in MiLEST are described in the next section.

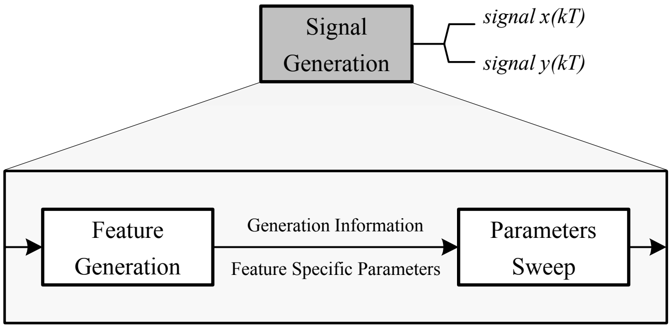


Figure 4.4: Signal-Features Generation – a Generic Pattern.

A similar approach is used for the *signal evaluation* (cf. Figure 4.5). Firstly, a signal is prepared for the extraction of a SigF of interest. This is called the preprocessing phase. Then, the feature

is extracted to be finally compared with the reference value. A verdict is set based on the applied arbitration mechanism. The details w.r.t. extraction of the concrete SigFs are given in Section 4.2, whereas the patterns classification leading to the test architecture and the arbitration mechanism are elaborated in the upcoming chapter.
The *time step size* employed in the process of signal-features extraction is a critical factor in establishing its success. The choice of the *time step size* is dependent on the different rates of response that the system exhibits. If it is chosen too small, it may result in a lack of sensitivity to changes: too large – it may produce incorrect inferences. Decreasing the time step may help in differentiating between discontinuities, abrupt changes, and continuous effects. On the other hand, if the time step is too small when applied to a variable with a relatively slowly decreasing slope, it appears that the signal does not change for a period of time; therefore, it is reported to be normal or to have reached steady state. In reality it is decreasing, and reporting it as normal may result in premature elimination of true faults [Mos97].

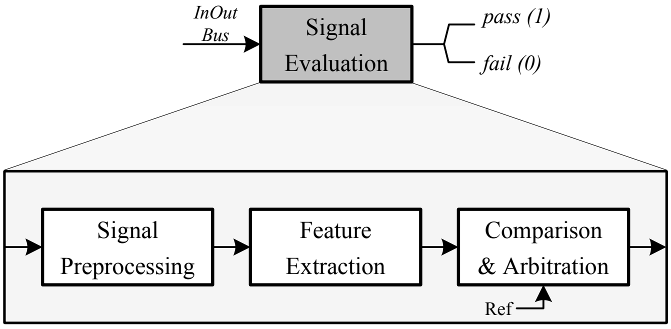


Figure 4.5: Signal-Features Evaluation – a Generic Pattern.

The concept of feature extraction and features classification were already given in the previous work of the author [ZSM06]. Then, the types of features discussed in this thesis and the substantial parts of the evaluation implementation are directly adopted from [MP07, ZMS07a].

Extracting SigF from a signal can be generally seen as a transformation of SUT signals to so-called *feature signals* (not to be confused with *signal feature*, also called SigF or simply feature in the following). The concrete values of an extracted *feature signal* represent the considered SigF at every time step. The *feature signal* is then compared with the reference data according to a specific, SigF related algorithm.

The scope of the online evaluation is reduced to all the past time steps until the actual one. Many features are not immediately identifiable, though. Taking an example of a maximum, it can only be detected at least one time step after it takes place. Some features are only identifiable with a delay, that might be known in advance (determinate) or not (indeterminate). This phenomenon is revealed in the naming convention for SigFs applied in this thesis.
The feature extraction realization determines the two aspects according to which the features can be classified. In Figure 4.6, one example for every combination is drawn, including the actual signal and the *feature signal* (i.e., result of the feature extraction). For triggered features an additional *trigger signal* exists.

Vertically, the classification in Figure 4.6 addresses the SigF identification trigger. Time-independent (also called non-triggered) features are identifiable at every time step, while triggered features are only available at certain time steps. In Figure 4.6, the valid evaluation time steps of the triggered features are colored in light green.

Horizontally, Figure 4.6 presents the identification delay, differentiating between no delay (immediately identifiable), determinate delay, and indeterminate delay. Immediately identifiable features are a special case of the features identifiable with determinate delay, but their delay equals zero. The signal value and the searched time step of a given signal are the immediately identifiable features in Figure 4.6, cases – a and b. In the latter, the *trigger signal* activates the comparison mechanism, whereas, the *feature signal* represents the simulation time. If a verdict for this check is being set, actually only three time steps deliver a verdict, for every trigger rise. Every conceivable causal feature can be classified under this aspect, i.e., all causal filter types, moving transforms, slope checks, cumulated values, etc.

When the identification of SigF occurs with a determinate delay (cf. Figure 4.6, cases c and d), the *feature signal* is delayed too. It reports about features in the past that could not be identified immediately. A prominent example is detecting a local maximum, for which a constant delay is necessary. The size of the delay varies depending on the maximum detection algorithm used. When the delay is constant and known, the time step when a certain feature value was observed can be determined. However, the signal evaluation is retarded. This fact is particularly important when considering relations between features in the upcoming sections. Other features identifiable with determinate delay are impulse detection algorithms or non-causal filters.

The features that cannot be identified immediately or with a determinate delay after the actual observation are exemplified in Figure 4.6f. There the rise time of a step response of a control loop is extracted. This feature is clearly triggered and the delay is indeterminate because it depends on the time when the actual loop will respond. Assuming that the step time is the observation starting point, the delay is then computed as the difference between the trigger rise and the observation time. This situation is indicated by the *reset signal*, in Figure 4.6f – the step time). The actual rise time (i.e., *feature signal* in this context) must be extracted not later than when the feature is triggered. In the figure, the rise time is available very early, but the test system gets this information when the feature is triggered. Other triggered features identifiable with indeterminate delay are, e.g., any other step response characteristic values, a system response delay, or the pattern complete step. Generally, most features based on the detection of two or more asynchronous events are of this type.

Finally, Figure 4.6e presents the maximal delay to date. This feature measures the delay between the actual SUT signal and a reference signal when the reference outputs a rising edge. Then, it compares the gathered value with the highest delay to date as soon as possible. When exactly this will happen is unknown in advance, though. Finally, the highest value is saved for the next simulation step. This feature is defined for every time step, although value changes are triggered. As expected, the *feature signal* is a stair step signal that can only increase. An implementation of this kind of feature extraction is not considered in this work since only a few features of this type could be identified so far and all of them were either describable using the formalism for triggered features or were of low practical interest.

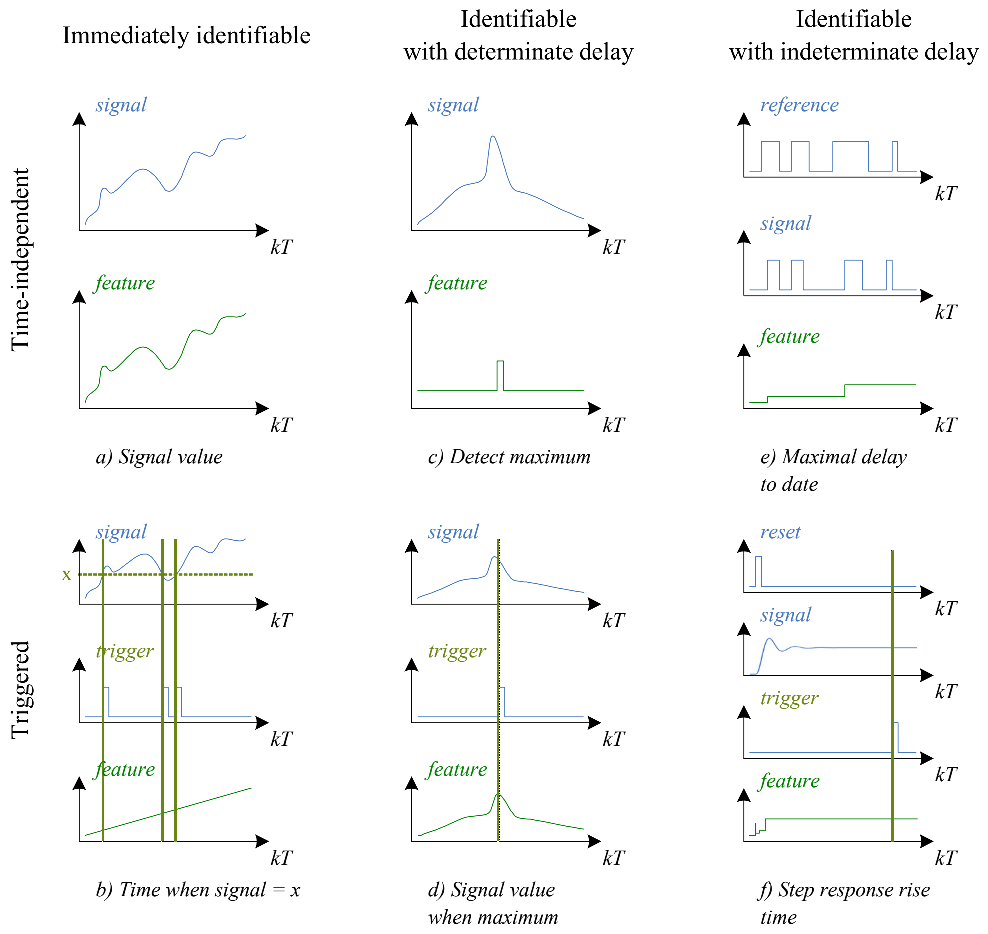


Figure 4.6: Signal-Features Classification based on the Feature Identification Mechanism.

The classification of different feature types given in Figure 4.6 is comprehensive as far as the output signals of a casual system are evaluated. Using the presented mechanisms, SigFs that are observable in the past up to the current time step, can be identified and assessed.
Under these circumstances, the classification of feature types is completed. Though, the instances representing those types may lead to inconsistencies, as Gödel's theorem [Fra05] would imply. Indeed, the different types of SigFs may often be implemented using different mechanisms depending on the current needs of the test system (cf. Figure 5.10).

The realization of the test evaluation enables to run it online, i.e., during the execution of the SUT. This implies that the feature extraction algorithm is run cyclically in SL. Hence, verdicts are computed at every time step on the fly.

In the next three sections, the three types of SigFs are explained in detail. The scheme of description is always the same. Firstly, the definitions are given. Then, a few generic examples are

provided, on the basis of which feature generation principles follow. They are listed in tables. Furthermore, the implementation examples are described in depth. In particular, algorithms for feature extraction are reviewed and the mechanisms for feature generation are discussed.
The separation into different feature types is motivated mainly by the fact that several features are not available at every time step and not all features can be extracted causally. For the time steps when the feature is not available, a *none* verdict is set.

4.2.2 Non-Triggered Features

Time-independent (non-triggered) features identifiable with or without a delay (TI) are available to be extracted at every time step. Thus, they can be described using a single *feature signal*. The generation of SigF simply produces a signal including this SigF. The SigF extraction is an algorithm that computes the actual value of the *feature signal* at every time step.

Table 4.1 outlines the examples of detection and generation algorithms for features classified as TI identifiable immediately or with a determined delay. The list is obviously not exhaustive. Only a set of basic examples are given based on the analysis of mathematical functions or features included in [Men97, MSF05, LKK$^+$06, SZ06, MathSL, WG07]. They enable, however, creation of more comprehensive features. In the further part of this section the details regarding the realization of the features generation and evaluation, both being the contribution of this thesis are explained.

Both activities – feature extraction and feature generation are parameterized. Since the signals generation occurs automatically (cf. Sections 5.3 – 5.5) additional issues on their derivation are given explicitly. These are generation information and a set of parameters strongly related to the extraction of features. These are of particular importance since they must be set on exactly the same values in the feature generation as in the respective feature extraction blocks. The parameters for the feature extraction may be found in the MiLEST library, instead.

Table 4.1: TI Features – Evaluation and Generation Algorithms.

SigF	***Evaluation View***	***Generation View***
	Time-independent (TI)	
Immediately identifiable	1-1. Signal value detection [LKK$^+$06]	1-1. Any curve crossing the value of interest in the permitted range of values, where duration time = default Generation information: – value of interest
	1-2. Basic mathematical operations, e.g., zero detection [MathSL]	1-2. Any curve described by a basic mathematical operations, e.g., crossing zero value in the permitted range of values, where duration time = default Generation information: – time of zero crossing
	1-3. Increase detection [MathSL, LKK$^+$06]	1-3. Any ramp increasing with a default/given slope in the permitted range of values, where duration time = default Generation information: – slope – initial output – final output
	1-4. Decrease detection [MathSL, LKK$^+$06]	1-4. Any ramp decreasing with a default/given slope in the permitted range of values, where duration time = default Generation information: – slope – initial output – final output
	1-5. Constant detection [MathSL, LKK$^+$06]	1-5. Any constant in the permitted range of values, where duration time = default Generation information: – constant value
	1-6. Signal continuity detection [Men97]	1-6. Any continuous curve in the permitted range of values, where duration time = default
	1-7. Derivative continuity detection [Men97]	1-7. Any continuous curve in the permitted range of values, where duration time = default
	1-8. Linearity (with respect to the first value) detection	1-8. Any linear function in the permitted range of values, e.g., described by the equation y =ax+b, where duration time = default Generation information: – slope – y-intersept Parameter: – linearity constant
	1-9. Functional relation y = f(x) detection	1-9. Any function in the permitted range of values described by a concrete y=f(x), where duration time = default
	1-10. Maximum to date detection	1-10. Any curve containing at least one maximum in the permitted range of values, where duration time is at least the time of date Generation information: – time of date
	1-11. Minimum to date detection	1-11. Any curve containing at least one minimum in the permitted range of values, where duration time is at least the time of date Generation information: – time of date
	1-12. Causal filters and moving transformations [SZ06]	1-12. Application specific
	1-13. Actual/cumulated emissions, consumption [SZ06]	1-13. Application specific

Identifiable with determined delay	2-1. Detection of local maximum [LKK+06]	2-1. Increasing and decreasing ramps one after another forming a maximum in the permitted range of values, where duration time = default Generation information: – duration of the increase and decrease – limits of the ramps in value range Parameter: – delay value while maximum detection
	2-2. Detection of local minimum [LKK+06]	2-2. Decreasing and increasing ramps one after another forming a minimum in the permitted range of values, where duration time = default Generation information: – duration of the increase and decrease – limits of the ramps in value range Parameter: – delay value while minimum detection
	2-3. Detection of inflection point [MSF05]	2-3. A curve, containing a point at which the tangent crosses this curve itself e.g., a curve $y = x^3$ forming an inflection point at point (0,0), where duration time = default Generation information: – inflection point Parameter: – delay value while inflection point detection
	2-4. Peak detection [MSF05]	2-4. Decreasing and increasing ramps one after another forming a peak in the permitted range of values, where duration time = default Generation information: – duration of the increase and decrease – limits of the ramps in value range Parameters: – minimal peak size – peak sensibility – delay value while peak detection
	2-5. Impulse detection [MSF05]	2-5. An impulse signal with a given impulse size and impulse duration in the permitted range of values, where duration time = default Generation information: – impulse size – impulse duration Parameters: – minimal impulse size – delay value while impulse detection
	2-6. Step detection [LKK+06]	2-6. Any step in the permitted range of values, where duration time = default Generation information: – step time – initial value – step size Parameters: – minimal step size – constant duration before a step
	2-7. Non-causal filters and moving transformations [SZ06]	2-7. Application specific

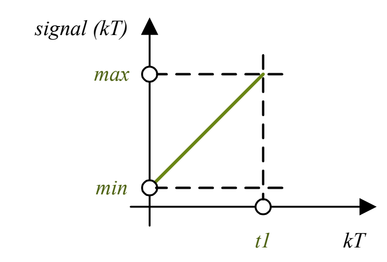
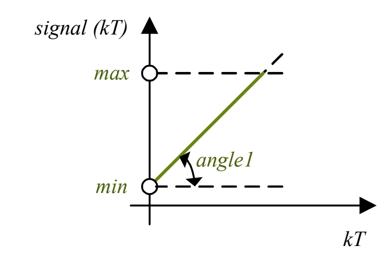
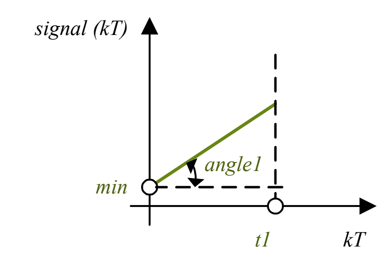
In the upcoming paragraphs, a further explanation of the selected TI features is presented and the SL implementation of their generation and extraction is shown.
By now, only the generation of single features is provided, omitting the concepts for adjusting their duration, their sequencing, or variants generation. These issues will be addressed in Sections 5.3 – 5.5.

Thereby, considering the *signal value* feature, its generation is reduced to producing any curve which crosses the value of interest within a predefined duration time. For *signal value* detection, the role of feature extraction block is to pass signal values to the comparison block. More complex features can be extracted applying the mathematical functionality of the standard SL libraries.

For generation of *increase, decrease,* or a *constant* various solutions are possible, e.g., ramps, logarithmic, or exponential functions. The features can be detected by analyzing their derivative. This can be approximated using the actual signal value and the past one (backward difference):

$$f(kT) = sign\ [signal\ (kT) - signal\ ((k-1) \cdot T)] \tag{4.1}$$

The *feature signal f(kT)* is 1 if the signal increases, 0 if it is constant, and −1 if it decreases. Similarly, the *continuity* of a signal can be checked by the argument of the sign function. If the backward difference exceeds a certain value, the existence of a step is assumed and a discontinuity is detected. The *continuity of the derivative* can be extracted in the same manner [MP07]. In the proposed approach, the realization of an *increase* is a simple ramp, of which two types are possible. One for an increase limited in time and in signal value range, the other for a change with a given slope and limited in value range. A simplified version of this feature generation algorithm is shown in Figure 4.7, including masks for the corresponding blocks in SL. Option 3 is not covered in the prototype.

no.	*Options*		*Generated signals*
1	*Criteria:*		
	time	+	
	signal range	+	
	tangent (angle)		
2	*Criteria:*		
	time		
	signal range	+	
	tangent (angle)	+	
3	*Criteria:*		
	time	+	
	signal range		
	tangent (angle)	+	

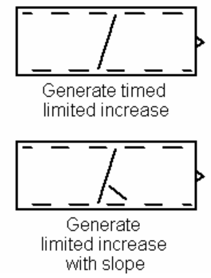

Generate timed limited increase
Generate limited increase with slope

Figure 4.7: Feature Generation: Increase Generation and the Corresponding SL Block masks.

In the first option the obtained ramp covers the entire range and preserves the proper time constraint (t_l). In the second option, the signal variants must hold for the given tan(angle) along the entire value range, no matter how long. Finally, in the third option only one default boundary of the *signal range* is considered, the *tan(angle)* is preserved and the predefined t_l indicates the duration of signal generation.

For *increase* detection the system in Figure 4.8 checks if the actual signal value is higher than the previous value. The clock and the switch were introduced to prevent bad outputs derived from an unfortunate choice of the initial output of the memory block. They provide no effect after the first time step.

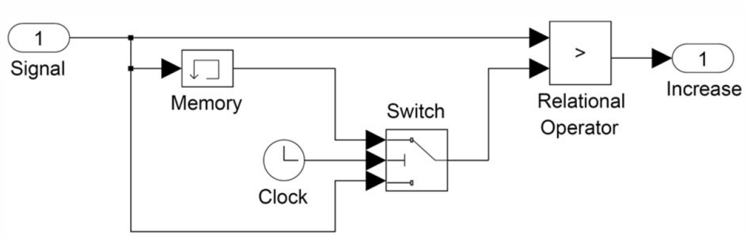


Figure 4.8: Feature Extraction: Increase.

The analog algorithms are valid for *decrease*, with the appropriate adjustments. Generation of a constant is a trivial task and must be limited only by the duration time.

A similar construction to that for the *increase* extraction appears in the implementation of *constant* detection in Figure 4.9, but in this case the switch is used to force a zero in the output in the first simulation time step. Additionally, the simple signal comparison block has been replaced by a subsystem implementing a signal comparison with a relative tolerance.

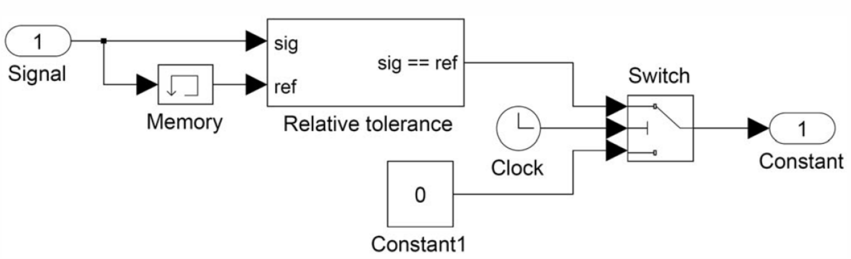


Figure 4.9: Feature Extraction: Constant.

In many cases it is necessary to allow for a small deviation when comparing signals. The reason is to avoid simple numerical precision faults. Differences between two signals in the tenth decimal place are often negligible and must be filtered out. Additionally, when using measured signals for stimulating the test system, deviations must also be considered and filtered out to some extent. This can be achieved, among other possibilities, by using the relative tolerance block, whose structure is shown in Figure 4.10. The constant value of *Relative Tolerance* indicates the targeted precision and is a user-defined parameter that can be set up in the feature extraction block mask.

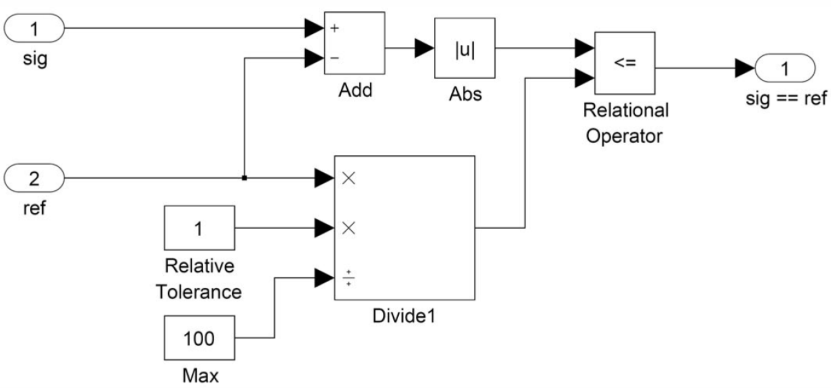


Figure 4.10: Relative Tolerance Block.

The *linear dependency of two signals* can be subsequently analyzed using the initial signal values (*signal (0)*) and storing the slope of the line crossing the actual values and the initial values at every time step. If the slope changes at some time step, there is a deviation from the linear behavior. Thus, the linearity is defined independently of the signal slope, with respect to the initial values and to the previous signal values. More sophisticated algorithms determining the linearity of two signals are obviously also possible.

A concrete *functional relation* between two signals can be easily realized with the mathematical SL functionality. As a matter of example, a *linear functional relation* between two signals is implemented. Its generation relies on the concept of a step (valued with *c*) starting at *0 time units* multiplied by the increasing time values as shown in Figure 4.11. The signals *u* and *y* are supposed to fulfill the relation *y* = *c·u*, where *c* is a feature parameter. Again, the equation is not implemented strictly, but using the relative tolerance block (cf. Figure 4.12).

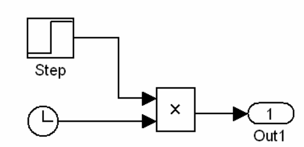


Figure 4.11: Feature Generation: Linear Functional Relation.

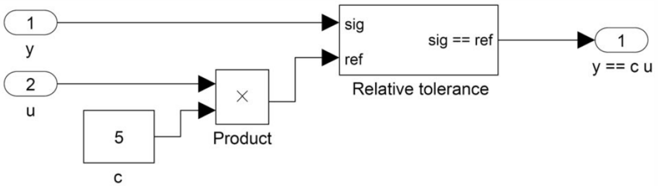


Figure 4.12: Feature Extraction: Linear Functional Relation.

The *maximum to date* and *minimum to date* functions can be generated applying simple repeating sequences and steps or sums of them in different combinations. The crucial issue is to note their duration time and synchronize them with the entire test case generation sequence; however, this problem will be discussed in Section 5.5.

The considered functions recursively compute the maximal or minimal signal value to date as follows:

$$max\ to\ date\ (kT) = max\ [max\ to\ date\ ((k-1)\cdot T)\ ,\ signal\ (kT)] \tag{4.2}$$

$$min\ to\ date\ (kT) = min\ [min\ to\ date\ ((k-1)\cdot T)\ ,\ signal\ (kT)] \tag{4.3}$$

Figure 4.13 shows the blocks implementing the *minimum to date* feature extraction. The last minimal value is saved in the memory block for the next time step.

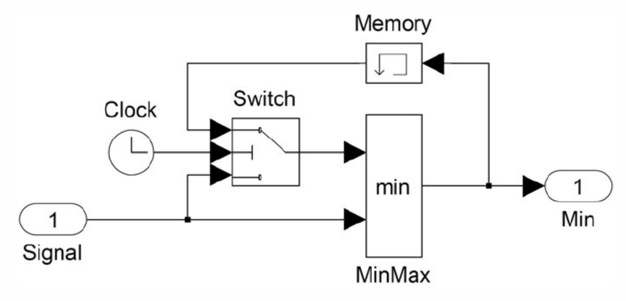


Figure 4.13: Feature Extraction: Minimum to Date.

The *local maximum* or *peaks* can also be easily produced using a repeating sequence function. The objective of *local maximum* detection is to check if the last backward difference was negative and the last before last positive. If that is the case, the *feature signal* is triggered, indicating the presence of a maximum with a time step delay, i.e., the maximum takes place one time step before it is actually noticed. The *feature signal* itself does not need to be retarded; it is already delayed.

A basic algorithm for detecting *local maxima* and *minima* starts from the value of the backward difference in the actual and next time step. By using the next time step, the feature extraction is delayed by one time step. If the backward difference changes its sign from positive to negative, there is a maximum, whereas a change from negative to positive points at a minimum. The situation is presented in Figure 4.14.

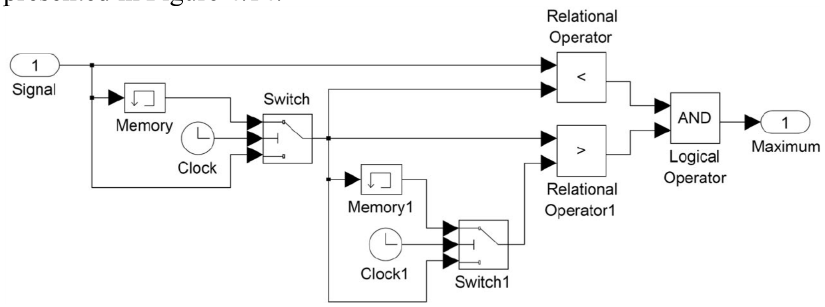


Figure 4.14: Feature Extraction: Local Maximum.

Inflection points can be detected similarly, since they are defined as maxima or minima of the derivative.

Discrete filters and *moving transforms* – such as the short-time Fourier transform (STFT) – produce a single value at every time step. Depending on the signal interval used for computing the actual *feature signal* value, the feature will be immediately identifiable (only past and actual signal values are used) or will be identifiable with a determinate delay (use of future values). *Peaks* are pronounced maxima or minima of short duration. They can be detected in a similar manner to maxima and minima, but the signal slope around the extremum should be of a minimal size.

The *peak detection* algorithm shown in Figure 4.15 is similar to the local maximum detection. This time, however, the last two backward differences d_1 and d_2 are explicitly computed. The feature parameter sensibility is user-configurable and determines the minimum step size before and after the peak. The product of both backward differences is used to assure that they have different signs. Since the peak is recognized only one time step after it actually takes place, the feature has a unit delay.

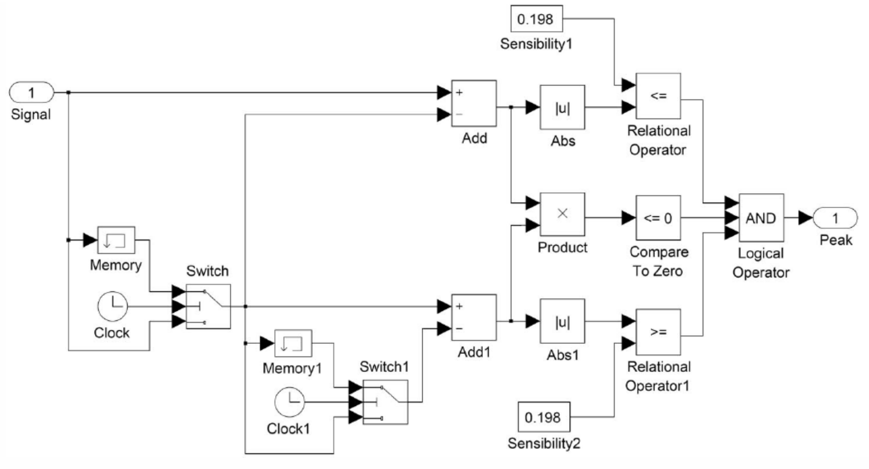


Figure 4.15: Feature Extraction: Peak.

Impulses are also TI features, since they are characterized by pronounced signal energy over a short time period. Computing the virtual *emissions* and *consumption* of automotive engines requires more complex algorithms, but these are TI features as well.

The realization of other feature types is partially based on the principles of TI features.

4.2.3 Triggered Features

Triggered features identifiable with determinate delay or without a delay (TDD) are basically TI features with special time constraints. They could be detected at every time step, indeed, but they are temporally constrained. As a consequence, they are not available at every time step. Modeling the extraction algorithm of these features implies gathering two pieces of information: the extracted SigF value itself and the time point or time range when the feature is valid.

Thereby, the generation of TDD feature requires information about the trigger and about the delay (if it exists).

In this context, the *feature signal* introduced for extraction of TI features is reused and completed by a *trigger signal.* The *trigger signal* indicates the time steps when the *feature signal* is valid. It can only take the values true and false. *Trigger* and *feature signal* should not be separated, since they do not contain enough meaning when isolated.

Normally, *feature* and *trigger signal* are computed separately, because the *trigger signal* represents usually the temporal constraints on the *feature signal*. Both are often independent. As already mentioned, this does not necessarily have to be the case, but it appears to be frequently that way. In consequence, the feature extraction problem for TDD features can be interpreted as the extraction of two separate TI features (cf. Section 4.2.1), resulting in *feature* and *trigger signal*.

In the same manner as for TI features, the examples of detection and generation algorithms for TDD features are listed in Table 4.2. Several of them are based on TI features with an additional time constraint.

Any immediately identifiable TI features over a time, e.g., signal value at time_1, signal value within [time_1, time_2] or any immediately identifiable TI features when a value is reached, e.g., time at value_1, time while maximum to date, increase within [value_1, value_2] belong to the category of TDD features.

Table 4.2: TDD Features – Evaluation and Generation Algorithms.

SigF	***Evaluation View***	***Generation View***
	Triggered (TDD)	
Immediately identifiable	3-1. Detection of signal value at a certain time step	3-1. Any curve crossing the value of interest in the permitted range of values, where duration time = default, but not less than a given certain time step Generation information: – signal value Parameter: – duration of the feature not less than the given time step
	3-2. Detection of time stamp of an event, i.e., time of an event	3-2. Any curve where an event appears in the permitted range of values, where duration time = default e.g., a signal flow consisting of increasing ramp, peak, increasing ramp, where duration time = default Generation information: – time stamp Parameter: – triggering value (e.g., event)
	3-3. Detection of increase rate	3-3. Any ramp increasing with a default/given slope in the permitted range of values, where duration time = default Generation information: – slope – initial output – final output
	3-4. Detection of decrease rate	3-4. Any ramp decreasing with a default/given slope in the permitted range of values, where duration time = default Generation information: – slope – initial output – final output
	3-5. Detection of the signal value when signal is constant	3-5. Constant in the permitted range of values, where duration time = default Generation information: – signal value
	3-6. Detection of time point since signal is constant	3-6. Constant following a non-constant curve in the permitted range of values, where duration time = default Generation information: – starting time of constant
	3-7. Step size detection [LKK+06]	3-7. Any step in the permitted range of values, where duration time = default Generation information: – step time – initial value – sample time size Parameters: – minimal step size – constant duration before a step
	3-8. Detection of functional relation in the first *t* seconds of the test	3-8. Any function in the permitted range of values described by a concrete y =f(x), where duration time is at least time of t seconds Generation information: – slope – y-intersept Parameter: – time *t*

Identifiable with determined delay	4-1. Value detection when TI feature identifiable with a delay is active, e.g., signal value when there is a local maximum	4-1. e.g., Increasing and decreasing ramps one after another forming a maximum in the permitted range of values, where duration time = default Generation information: – signal value Parameter: – delay value while maximum detection, i.e., here specific for the maximum detection algorithm
	4-2. Detection of time stamp for TI features identifiable with a delay	4-2. e.g., Increasing and decreasing ramps one after another forming a maximum in the permitted range of values, where duration time = default Generation information: – time stamp Parameter: – delay value
	4-3. Extraction of signal energy during an impulse [MSF05]	4-3. An impulse signal with a given impulse size and impulse duration in the permitted range of values, where duration time = default Generation information: – Signal energy Parameter: – delay value

Signal value at a certain time step is the simplest example of a TDD feature. From the extraction point of view, it is actually a TI feature with an additional *trigger signal* given by the explicit time constraint. The constraint can also be determined by an event that activates the trigger. Any signal including an event (e.g., signal value=*x1*) that appears at a given time *t1* satisfies the requirements for generation of this feature (cf. Figure 4.16).

Concerning the feature extraction in Figure 4.17 the actual time is continuously output, whereas the trigger becomes active only when an event occurs. In this case it happens when the input signal reaches a certain value. Both *trigger* and *feature signal* are computed separately by two different TI features completing each other.

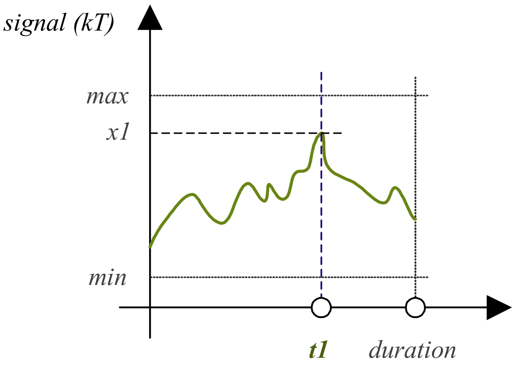


Figure 4.16: Feature Generation: Time Stamp of an Event.

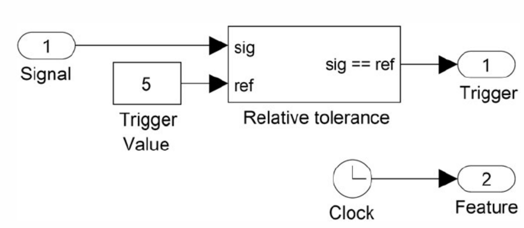


Figure 4.17: Feature Extraction: Time Stamp of an Event.

In the implementation presented in Figure 4.17, the relative tolerance block has been used. It helps avoid passing the Trigger Value without triggering due to the limited temporal resolution of the signal, but it introduces new problems as well: the trigger could be active more than once if the tolerance is not optimally configured. Again, the algorithm is not a real-world solution; it needs to be improved in order to work reasonably well. But it reproduces the basic principle appropriately.

The *increase rate* of a signal is another possible TDD feature, since it is only available when the signal increases. Hence, the trigger becomes active only when the signal increases. The *feature signal* is computed independently of that, outputting the backward difference. The backward difference represents the feature extracted when it is positive. The *decrease rate* and the *signal value when constant* can be extracted analogously.

The generation of feature *time since a signal is constant* is presented in Figure 4.18. A non-constant curve is followed by a constant starting at a given time step. Hence, the starting time ST of a constant is required generation information. In the realization below, a sine wave is followed by a constant.
Figure 4.19 shows that the *trigger* and the *feature signal* do not necessarily have to be independently extracted. The *trigger signal* is active whenever the input signal is constant. The *feature signal* is a time counter that will be reset at the trigger rising edge.

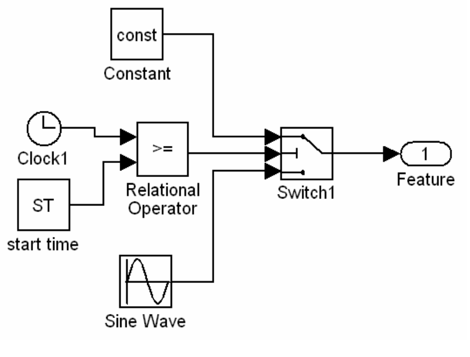


Figure 4.18: Feature Generation: Time since Signal is Constant.

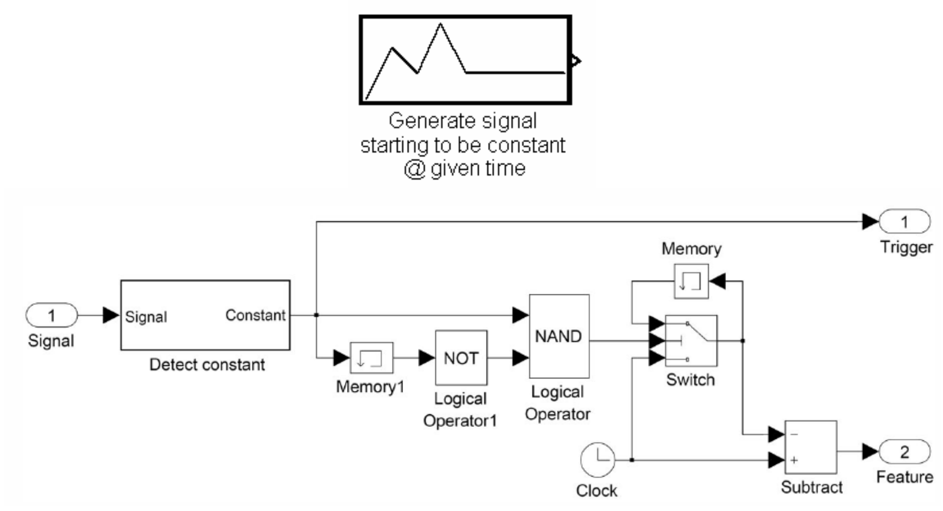


Figure 4.19: Feature Extraction: Time since Signal is Constant.

The extraction of the *step size* is derived from the detection of a TI feature, called *step* detection, as well as the functional relation in a certain time interval. The generation of a step is trivial applying the step generator from the standard SL library; however, it becomes interesting as far as the test reactiveness is considered. This particular case will be explained in Section 5.5 of this chapter and illustrated in the second case study, in Chapter 7.

Delays are also necessary when extracting triggered features and the delay is applied to both *trigger* and *feature signal*. All features that are based on a TI feature identifiable with determinate delay posses a delay automatically being the TDD feature. For example, the signal value of every local maximum of a signal is a TDD feature identifiable with a delay, since local maxima can only be identified with a delay. The implementation in SL illustrated in Figure 4.20 is comparatively easy using the TI feature maximum detection (presented in Figure 4.14). Both feature extraction outputs have a unit delay. Features extracting the time of an event may obviously appear with delay too.

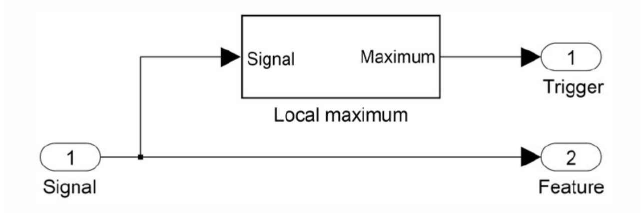


Figure 4.20: Feature Extraction: Signal Value at Maximum.

As already mentioned in Section 4.2.1 the generation of a *local maximum* can also be easily achieved using a repeating sequence function. A similar argumentation applies to the generation of TDD features involving a maximum. For illustration purpose a number of combinations is given in Figure 4.21. For all the listed instances a signal containing a maximum should be produced. However, each of them is characterized by different properties. Hence, for the feature called *Time Stamp at Maximum=Value*, a signal including the maximum of a given value at an appropriate time point is generated; for *Time Stamp at Maximum with given Slope*, a signal including the maximum determined by a ramp with a given slope value at an appropriate time point is obtained; for *Time Stamp at Maximum,* simply a signal including the maximum at an appropriate time point is provided; and for *Signal Value at Maximum* a signal including the maximum of a given value appears.

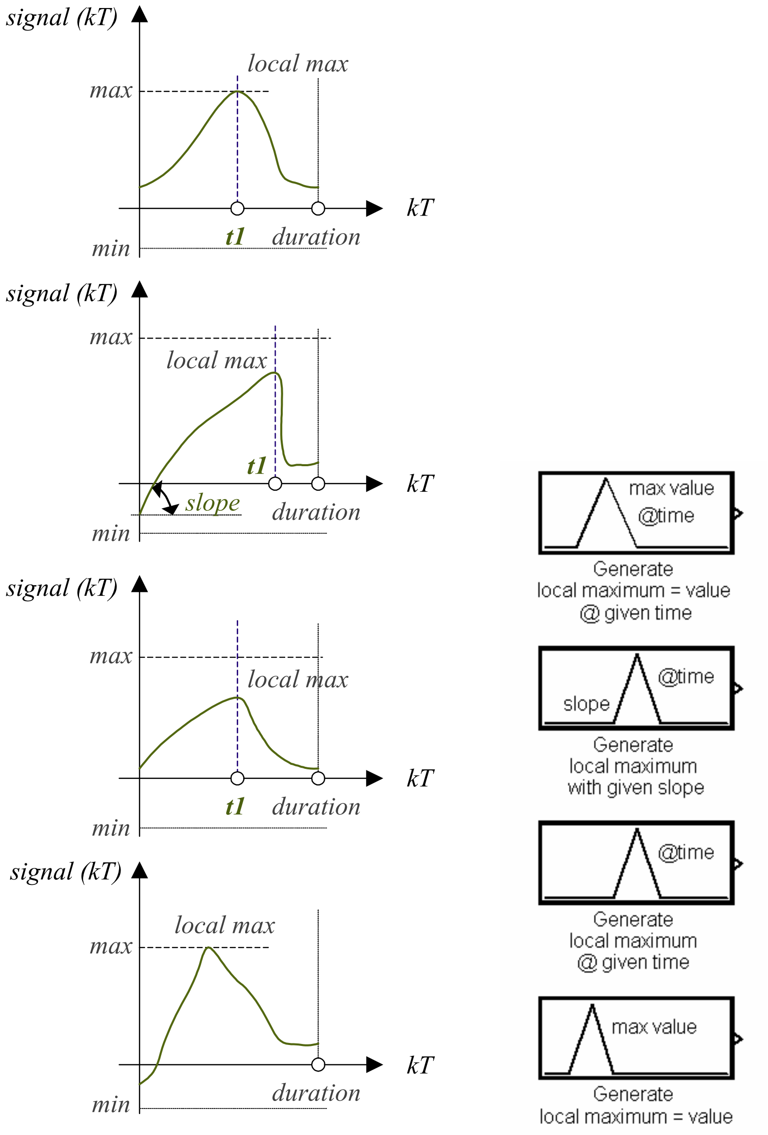


Figure 4.21: Examples for Generation of TDD Features Related to Maximum and their Corresponding SL Block Masks:
Time Stamp at Maximum = Value,
Time Stamp at Maximum with given Slope,
Time Stamp at Maximum,
Signal Value at Maximum.

Summarizing, TDD features extraction, in comparison to TI features extraction, needs second signal, called *trigger signal* which is necessary for their description. This additional signal only takes on Boolean values, but both signals are extracted in a similar manner.

4.2.4 Triggered Features Identifiable with Indeterminate Delay

Triggered features identifiable with indeterminate delay (TID) are available only with an indeterminate delay. As a consequence, they are distributed over a previously unknown number of simulation iterations. In other words, a single feature extraction algorithm can require different amounts of time depending on the SUT behavior. Thereby, the generation of TID feature requires information about the triggers needed for the extraction and the delay of the feature. This behavior contrasts with TI and TDD features presented so far. TDD features are not always available, but they are identifiable with a determinate delay or even without a delay. However, this fact implies that they are computable at every time step. The algorithm runs cyclically and extracts a feature in predefined time frames.
TID features are extracted sequentially, under the assumption that the same features do not overlap in the signal. The extraction implementation of TID features is based on the extraction of three signals: characteristic feature value (*feature signal*) – as already discussed for TI features, time when the feature value is available (*trigger signal*) – as added for TDD features, and the observation point, establishing the time range when the feature is valid (*reset signal*).

The *feature signal* represents the values of the extracted feature in time; however, its value is considered only when the *trigger signal* is active. The *reset signal* monitors the feature extraction process and becomes active for one single time step when the feature extraction is completed. It indicates the delay of TID features and can have a value of true or false – the same as the *trigger signal*.

The *reset signal* can be obtained in a similar manner to the *trigger* and *feature signals*. However, it is allowed to become active not later than the trigger. Hence, either both signals become active at the same time or the *reset signal* is followed by the activation of the *trigger signal*. Otherwise, the trigger activation is ignored.

The examples of detection and generation mechanisms for triggered features identifiable with indeterminate delay are outlined in Table 4.3.

Table 4.3: TID Features – Evaluation and Generation Algorithms.

SigF	*Evaluation View*	*Generation View*
	Triggered (TID)	
Identifiable with undetermined delay	5-1. Detection of time between two events	5-1. Any non-constant curve where two concrete events appear one after another in the permitted range of values within the given time range Generation information: – time of event$_1$ (*t1*) – time of event$_2$ (*t2*) not exceeding the permitted duration, where *t1<t2*
	5-2. Detection of signal mean value in the interval between two events	5-2. Any non-constant curve intersected by two concrete events sequenced one after another in the permitted range of values within the given time range Generation information: – signal mean value
	5-3. Detection of response delay [MSF05]	5-3. A stabilized constant followed by a step response characteristics with given response delay in the permitted range of values within the given time range Generation information: – response delay
	5-4. Complete step detection [MSF05, LLK^{+}06]	5-4. At least two steps one after another starting at a default/given value with a default/given step size and a default/given time between them and in the permitted range of values within the given time range Generation information: – step size – time between steps Parameters: – constant duration before a step – minimal step size
	5-5. Detection of step response characteristics [LLK^{+}06], e.g., – steady-state error – rise time – overshoot – settling time	5-5. A stabilized constant followed by a step response characteristics in the permitted range of values and time Generation information: – steady-state error – rise time – overshoot – settling time Parameters: – constant duration before a step – minimal step size – systems static gain – rise time interval lower limit (in set point %) – rise time interval upper limit (in set point %) – maximum overshoot - moving average weight – settling time range (in set point %) – steady-state error - moving average weight

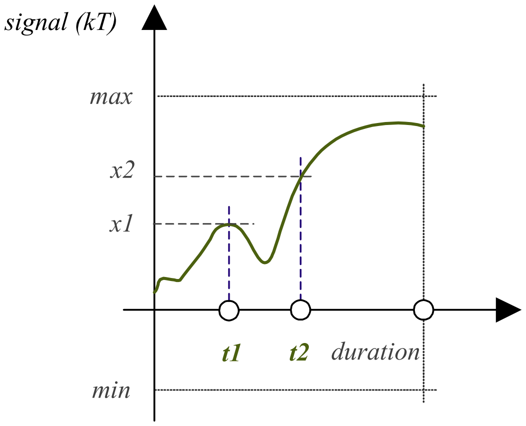


Figure 4.22: Feature Generation: Time between Two Events.

Time between two events is one of the simplest TID features. From the generation viewpoint, almost any non-constant curve in the permitted range of values and duration time fulfills the generation requirements as illustrated in Figure 4.22. The feature is obtained by projecting the times of events *t1* and *t2* onto the generated signal. These determine the values of the signal and are treated as events in this particular case. From the evaluation viewpoint, it is arbitrary when the TID feature appears – thus also when the events appear – but they must follow each other in a sequence, so as to catch the time between them.

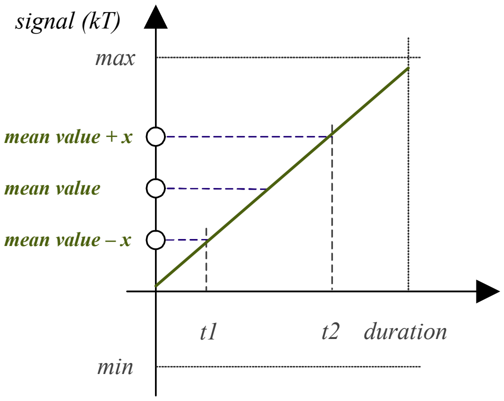


Figure 4.23: Feature Generation: Signal Mean Value in the Interval between Two Events.

Signal mean value in the interval between two events is based on the previous feature. Its generation algorithm is presented in Figure 4.23. Here again, almost any non-constant curve in the permitted range of values and duration time fulfills the generation requirements. However, the feature is obtained by projecting the mean value between two automatically determined values *mean value+x* and *mean value–x* onto the generated signal. By that, their occurrence times are also automatically determined. These points are treated as events in this particular case. Another possible generation algorithm would be to provide the events explicitly and manually on the curve by preserving the mean value of interest. Numerous other features alike and combinations of them are possible.

The *response delay* can be measured in a situation when an input step is applied on a stabilized system. The time needed for a system to respond is the delay. In the evaluation system the *reset signal* becomes active at the time of step occurrence – it being the decisive factor for any relations with other features.

A generation algorithm for a *complete step* will be described in Section 5.5 as it exploits the concept of test reactiveness. It can be extracted by analyzing a step and the signal right before the step, which must be constant. The situation becomes even more complex when a steady state of a system is considered. In that case, both SUT input and output signals should be constant for a minimum time, before the step eventually appears.

The detection of a *complete step* can be used to analyze the system *step response*, since it includes all preconditions necessary for the correct measurement of the step response characteristics. The step response is widely used for the description of the behavior of control systems.
A common *step response* of a second-order linear system[28] is drawn in Figure 4.24. Here four characteristics of the step response are marked. The upper plot shows the corresponding step signal causing the step response below.

In the following paragraphs, the step response characteristics will be defined, before some in-depth insight is given into the implementation of the actual feature generation and extraction.

Hence, the *rise time (t_r)* is defined as the time the system response needs to get from 10% to 90% of the set point y_{ss} after the step. Thus, a short rise time will mean a rapid system response to the new input situation. Shorter rise times are commonly associated with a larger *maximum overshoot*, i.e., the step response shoots over the actual target. For the maximum overshoot many different definitions are used. A widespread one defines it as a percentage of the set point as:

$$\frac{M_p - y_{ss}}{y_{ss}} \tag{4.4}$$

where M_p is the step response value at the maximum peak.

A further step response parameter is the *settling time*. It indicates how long it takes to leave the transient state and thus reach the steady state. In practical terms, this is the time between the input step and the last time point when the signal crosses into a user-defined tolerance range around the set point. Finally, the *steady-state error* provides information about the final deviation of the signal from the expected reference value *(r)* in the steady state. Before this deviation can be measured it must be assured that the steady state has been reached. Further step response characteristics such as the delay time or the peak time will not be considered in this work.

[28] Second-order linear systems are the simplest systems that exhibit oscillations and overshoot [Kuo03].

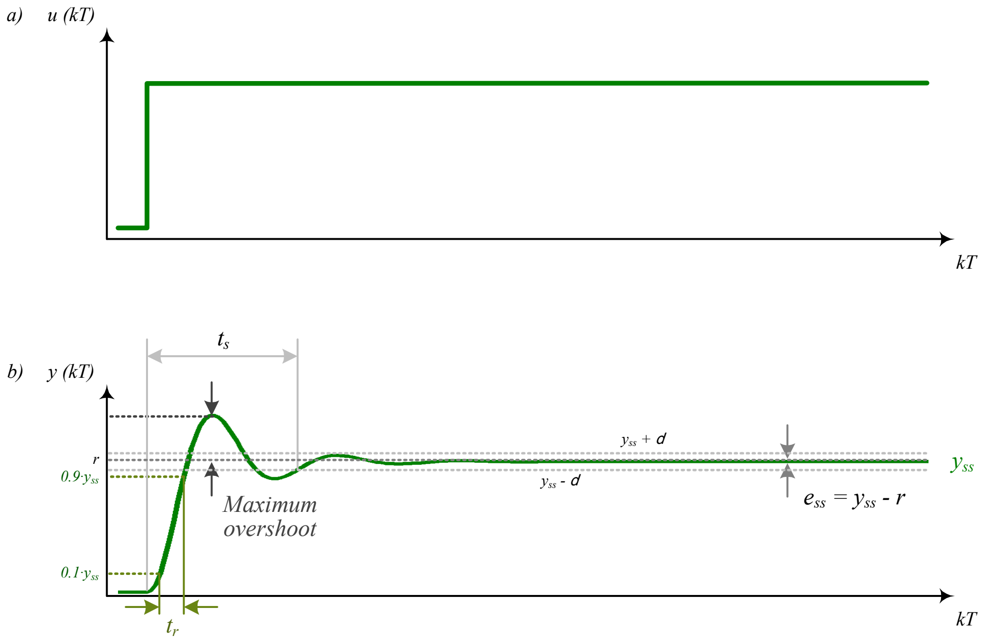


Figure 4.24: Reaction on a Step Function:
a) A Step Function – u(kT).
b) Step Response Characteristics y(kT): rise time (t_r), maximum overshoot, settling time (t_s) and steady-state error (e_{ss}).

Generation of a step response is of lower practical importance than its evaluation as usually the controller outputs are to be checked and not produced. However, a proposal for a simple generation will be introduced for reasons of completeness. It is realized as a MATLAB (ML) script. It is based on adjusting the damping ratio and natural frequency[29] assuming that one unit step is applied at the input.

The simplest second-order system satisfies a differential equation of this form [Kuo03, EMC⁺99]:

$$\frac{d^2y}{dt^2} + 2\zeta\omega_n \frac{dy}{dt} + \omega_n{}^2 y = G_{DC}\omega_n{}^2 u\ (t) \qquad (4.5)$$

where:

- *y(t)* – response of the system
- *u(t)* – input to the system
- ζ – damping ratio
- G_{DC} – DC (direct current) gain of the system
- ω_n – undamped natural frequency

[29] A normalized step response computed as a result of adjustment of the considered parameters can be animated by accessing the dedicated web page of University of Hagen: http://www.fernuni-hagen.de/LGES/playground/miscApplets/ Sprungantwort.html [04/30/2008].

The parameters determine different aspects of various kinds of responses. Whenever an impulse response, step response, or response to other inputs is concerned, the following relations apply [Kuo03, EMC+99]:

- ω_n will determine how fast the system oscillates during any transient response
- ζ will determine how much the system oscillates as the response decays towards steady state
- G_{dc} will determine the size of steady-state response when the input settles out to a constant value.

Deriving the response $y_s(t)$ to a step of unit amplitude, the forced differential equation is:

$$\frac{d^2 y_s}{dt^2} + 2\zeta\omega_n \frac{dy_s}{dt} + \omega_n{}^2 y_s = u_s(t) \tag{4.6}$$

where $u_s(t)$ is the unit step function.

To illustrate, the solution obtained for the equation (4.6), where $\zeta = 1$, is:

$$y_s(t) = \frac{1}{\omega_n{}^2}\left[1 - e^{-\omega_n t} - \omega_n t e^{-\omega_n t}\right] \qquad for \quad \zeta = 1. \tag{4.7}$$

The second-order system step response is a function of both the system damping ratio ζ and the undamped natural frequency ω_n. For damping ratios less than one, the solutions are oscillatory and overshoot the steady-state response. In the limiting case of zero damping the solution oscillates continuously about the steady-state solution y_{ss} with a maximum value of $y_{max} = 2y_{ss}$ and a minimum value of $y_{min} = 0$, at a frequency equal to the undamped natural frequency ω_n. As the damping is increased, the amplitude of the overshoot in the response decreases, until at critical damping $\zeta = 1$, the response reaches steady-state with no overshoot. For damping ratios greater than unity, the response exhibits no overshoot, and as the damping ratio is further increased the response approaches the steady-state value more slowly.

Manipulating the damping ratio and natural frequency enables different graphs with various characteristics to be obtained. The possible step responses of stable second-order systems are plotted in Figure 4.25 in terms of non-dimensional time $\omega_n t$ and normalized amplitude $y(t)=y_{ss}$.

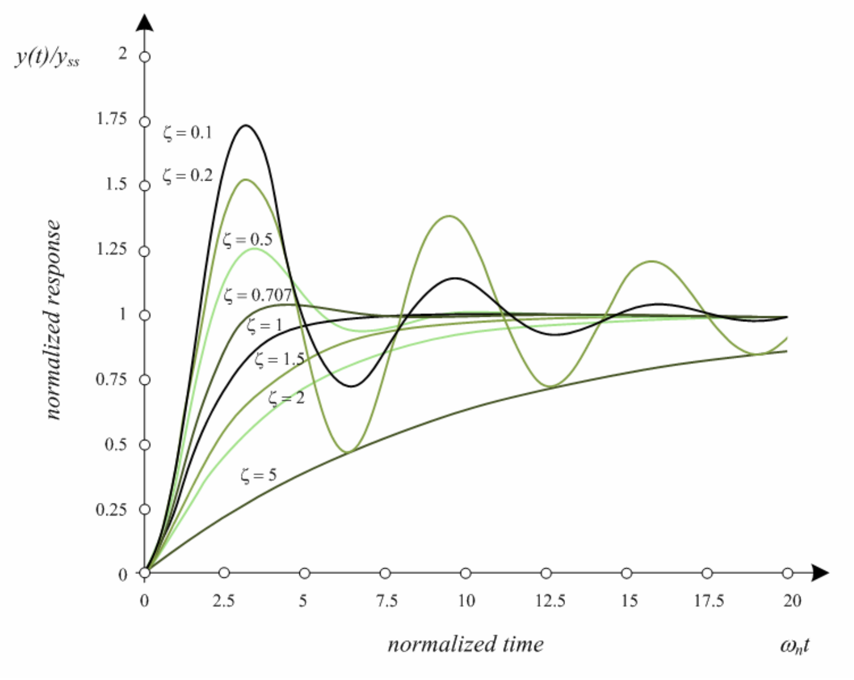


Figure 4.25: Step Response of Stable Second-Order System for Different Damping Ratios.

Then, the realization of an online extraction algorithm for the step response characteristics in SL is a more complex task due to their dependency on the input signal. It has been originally proposed by [MP07]. Starting with the extraction of the newly introduced *reset signal* the implementation is presented in Figure 4.26. The *reset signal* is common to all four considered features and it becomes active whenever a step appears under the condition that SUT input and output have been constant for some time. The memory block in Figure 4.26 is necessary to delay the result of this extraction by one unit.

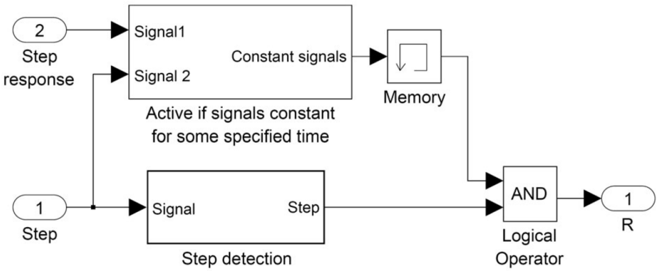


Figure 4.26: Reset Signal Extraction for Step Response Evaluation.

The diagram checking if both signals are constant for some user-specified time is shown in Figure 4.27. The TDD feature extraction block time since signal constant has been utilized twice; the minimum of both TDD features delivers the time since both signals were constant. If the triggers are true, the time value is extracted and compared with the minimum constant time. The latter is specified for the reset signal extraction.

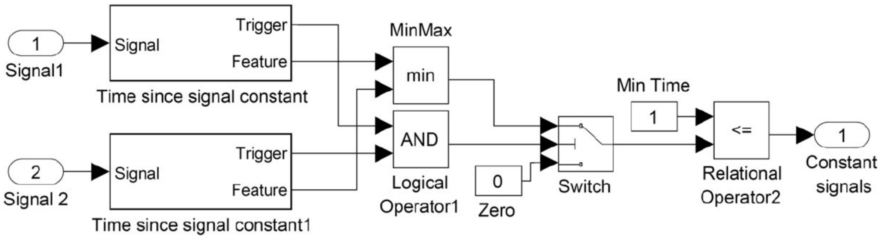


Figure 4.27: Constancy Check for a Given Minimal Time within the Reset Signal Extraction.

Figure 4.28 contains the diagram of the step detection algorithm, including a further parameter – the minimal step size – which must also be set while generating the step and the step response.

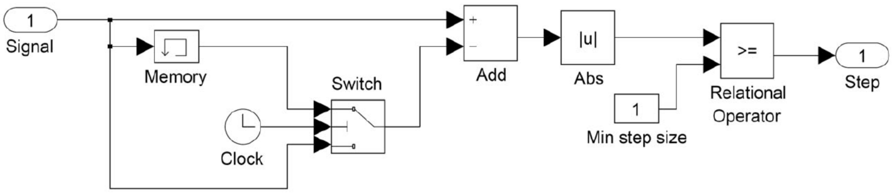


Figure 4.28: Step Detection within the Reset Signal Extraction.

An additional parameter – the relative tolerance value – is hidden behind the constant detection algorithm. It must be provided for both generation and extraction of step and step response. The parameters can be set up in the mask GUIs of the corresponding blocks.

The *feature signal* extraction for the four considered SigFs is more complex. They are checked in parallel so as to take advantage of synergies during their extraction.

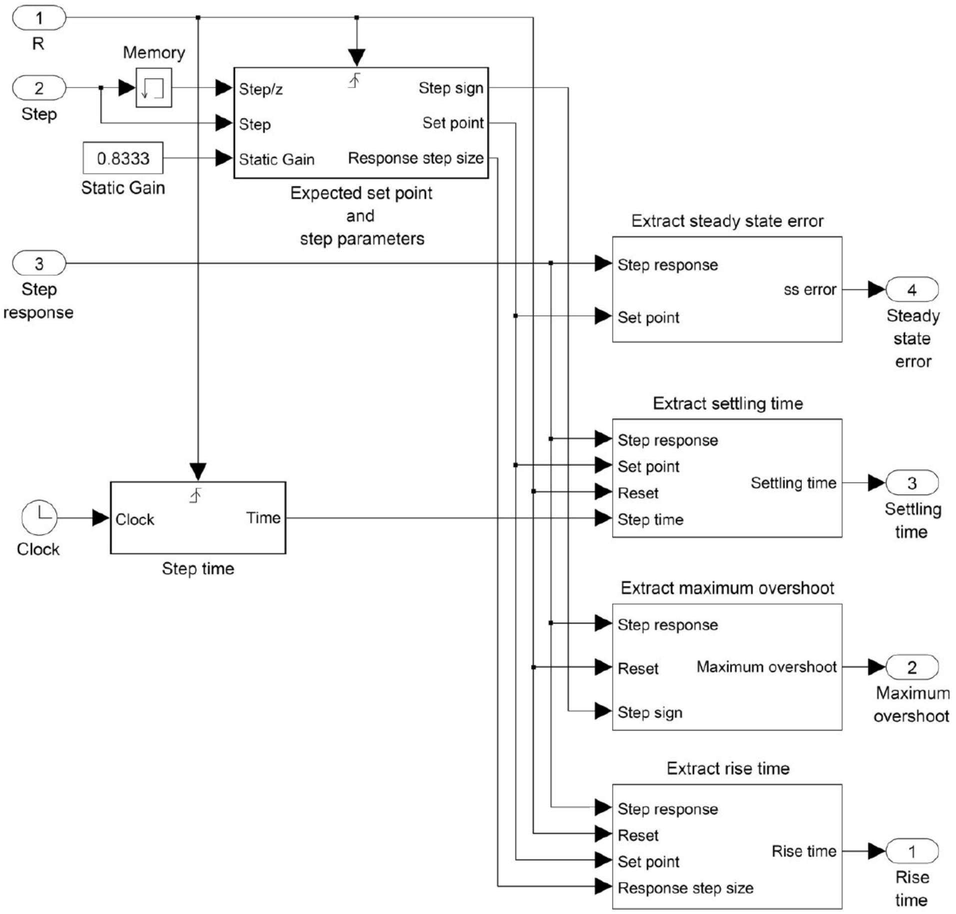


Figure 4.29: Feature Signals Extraction for Step Response Evaluation.

Figure 4.29 shows the insights of *feature signals* extraction of the step response characteristics. For each of them a separated block is provided. Additionally, the time of step occurrence (step time) is computed using a triggered subsystem that is activated by the *reset signal* and thus holds the step time.

The feature extraction algorithms of the rise and settling time need to know the set point y_{ss} in advance. Instead of y_{ss} the reference value r is used, because the y_{ss} is not available in advance. In the implementation, the reference r is designed as set point y_{ss} for simplification. The *response step size* – the difference between the set points after and before the step – and the step sign are calculated by the block called *Expected set point and step parameters,* the insights into which are shown in Figure 4.30. The expected set point is computed using a parameter – the static gain of the system.

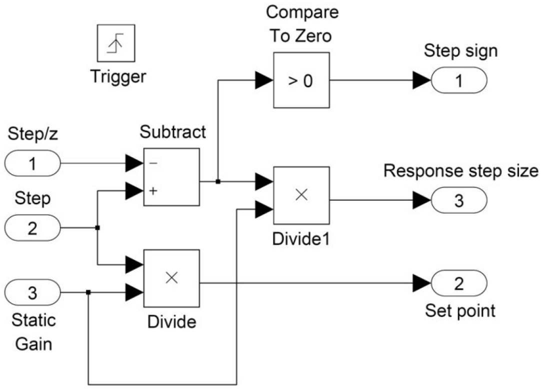


Figure 4.30: Computing Response Step Size, Step Size and Expected Set Point.

Figure 4.31 shows the implementation of the *rise time* extraction algorithm. When the step response crosses the 10% of *r* and then 90% of *r*, it triggers the subsystem which calculates the time difference between two last times it was activated. The activation is an effect of the signal from XOR block or the reset signal.

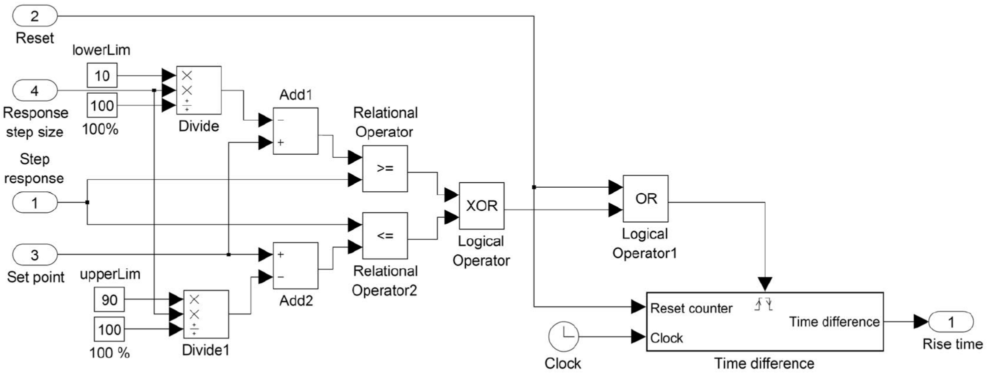


Figure 4.31: Feature Extraction: Rise Time.

If the *reset signal* does not become active during the rise time (i.e., no new step appears), the algorithm measures the rise time and holds it at the output. The assumption is, however, that the signal will not go back to the value of 90% of *r*, which can happen in reality. Hence, the implementation of the *Time difference* block is refined and shown in Figure 4.32. The two memory blocks on the right store the last two activation times, whereas the *Execution counter* block counts the execution times of the triggered subsystem. These are limited to 3 so as to catch the rise time limits properly. The counter is reset by the *reset signal* every time when a new step appears.

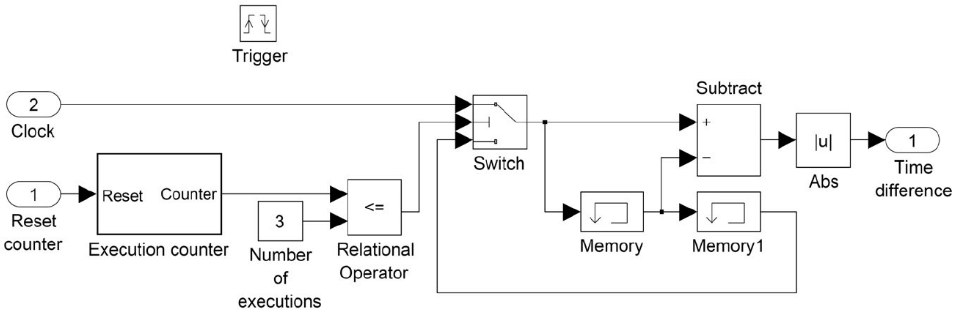


Figure 4.32: Time Difference Block for Rise Time Detection.

The *settling time* extraction algorithm is presented in Figure 4.33. Here, the time difference between the time point when the signal stabilizes and the time of step occurrence is computed. Every time the step response enters the tolerance range around the expected set point, a time stamp is made. If the tolerance range is not left any more, the settling time has been caught. This is the last time difference held before the trigger becomes active. Furthermore, the *reset signal* resets the triggered subsystem, called *Time stamp* at the beginning of every step, deleting any old information stored inside.

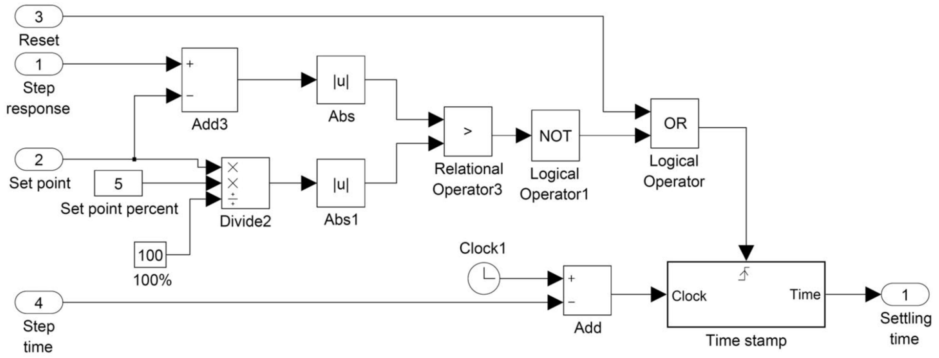


Figure 4.33: Feature Extraction: Settling Time.

The *steady-state error* is computed at every time step. The expected set point is subtracted from the actual signal value and the difference is filtered using a moving average block as shown in Figure 4.34.

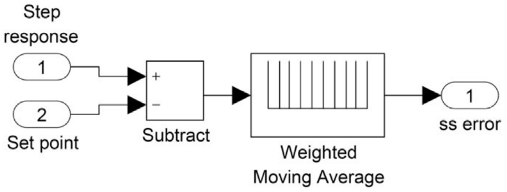


Figure 4.34: Feature Extraction: Steady-State Error.

The extraction of the *maximum overshoot* is more complicated and the implementation details are left out of the scope in this description. They can be found in the work of [MP07].

Finally, also the *trigger signal* must be computed so as to let the evaluation mechanism work properly. The trigger for the *steady-state error* is activated when the step response has stabilized and the SUT input signal has not changed its value after the step. These constraints also guard the termination of the extraction algorithms for the other three features. For checking them, the algorithm presented previously in Figure 4.27 is used in combination with the TI feature *detect increase* for detecting the rising step. With this practice, the trigger signal becomes active for only one time step. Additionally, the *steady-state error* should remain within a certain range so as to guarantee that the proper stabilization has been reached. The trigger algorithm is shown in Figure 4.35.

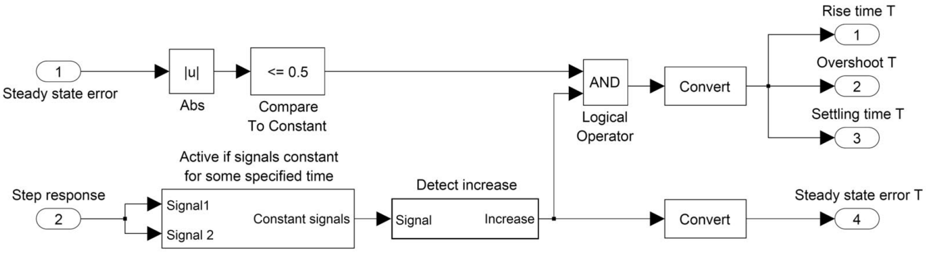


Figure 4.35: Extraction of Trigger Signals.

To sum up, generation of the TID features is relatively similar for TI and TDD features since only more parameters appear, making the implementation not exceptionally difficult. Their evaluation, however, needs three types of signals for the proper extraction: *feature*, *trigger* and *reset*. Such an approach reduces the complexity, which otherwise becomes large.

4.3 The Resulting Test Patterns

Test patterns [VS04] can effectively facilitate the automation, reusability, and maintenance of the test specification process if used appropriately. Hence, cost, time, and resources planned for the development of quality assured embedded systems decrease.
In this work the patterns for test harness, test data generation, test specification, SigF generators, SigF extractors, test evaluation, and test control are discussed. Their realization is provided in MiLEST library.

[Bert07] argues that any good testing practices need to be collected by a systematic effort so as to organize recurring effective solutions into a catalog of test patterns, similarly to what is now a well-established scheme for design approaches. Such a *test pattern extraction* [VS04] is the process of abstracting from existing problem-solution-benefit triples in the test developing process, to obtain patterns suitable for reuse in future contexts. Although the process of going through existing test artifacts and trying to identify patterns for later reuse might appear costly and unrewarding at first sight, it pays off in the long term [VS04]. This argumentation especially applies to the testing of embedded software, where reusability of some common test procedures is appreciated. The test patterns provide means for test system developers to focus more on what to test and less on the notation itself. With this practice, they simplify the test development process, increase the level of automation and facilitate the understandability of the test.

The patterns proposed in this thesis support both a manual and an automated test generation approach as [Neu04, PTD05] discuss. In a manual development approach, a developer can reuse patterns. An automated approach may benefit from an automatic identification of patterns so as to provide further solutions for the revealed issue [Neu04].

The already discussed features classification reveals that the features themselves follow some patterns. The realization of features generation and extraction constitutes the lowest abstraction level of the test data, test evaluation, and test oracle patterns. Further on, test control patterns are distinguished in the proposed approach. A test control is a specification for the invocation of test cases assuming that a concrete set of test cases within a given test configuration exist. Hence, the considered patterns can in fact be seen as parametrizable libraries. However, the customization possibility makes them more abstract than a library is in a traditional meaning. In [PTD05] a variant of patterns called *idiom* is considered. *Idioms* provide solutions for general problems arising using a certain executable programming language. In that sense, whenever ML/SL/SF is understood as a programming language, the test patterns presented in this thesis can be considered as such *idioms* because they are implemented in this language; however, they will be called *test patterns* here. All the mentioned test patterns present abstract solutions for generic problems. They are instantiated in MiLEST and summarized in Appendix B and will be explained in detail in further sections.

The test design patterns are provided in a graphical form. Thus, the textual table-form templates suggested by [Bin99] are not used. Instead, it is assumed that a short explanation of a graphical user interface (GUI) for each pattern block is informative enough to express its meaning, context and the application sense.

4.4 Test Development Process for the Proposed Approach

The signal generation and signal evaluation in isolation, no matter how sophisticated, neither test the system automatically nor make testing systematic or completed. These activities as such should be embedded in a clear and well-defined test process. Moreover, an appropriate framework has to deliver easy means for test specification, generation, and execution. For these reasons a method specific test development process is introduced in this section and a hierarchical architecture of the resulting test system in Chapter 5. Both are facilitated by application of test patterns in different constellations.

The origin of any test specification is the document that specifies the SUT requirements. Usually, they are available in a textual form. System requirements are often hierarchical, starting from high level, down to concrete technical specification. In some cases formalized versions are provided. From such textual documents, test requirements (here also called test objectives) should be derived. According to the general test process these can be classified as a set of the abstract[30] test scenarios. These test scenarios can be described by a conditional form which relates incoming SUT stimulation to the resulting SUT behavior by IF–THEN rules (4.8).

A technically-oriented MiLEST test development process proposed in this thesis is shown in Figure 4.36. The MBD paradigm assumes that the SUT model is already available and that the input/output interfaces are clearly defined and accessible.
Besides the analysis of the SUT specification, a proper functional, dynamic testing also requires a systematic selection of test stimuli, appropriate test evaluation algorithms, and obviously a test execution or simulation environment. Thereby, if the above assumptions hold, pattern for the generation of test harness[31] model can be applied to the SUT model as denoted by *step I* in Figure 4.36. This is done automatically with a MiLEST transformation function, giving an abstract frame for test specification. This phase together with the definition of IF-THEN rules is called the test design. Further on, the test specification and test implementation phase is carried out in *step II*, where the test definition in MiLEST is concretized based on the test requirements. The test engineer manually refines the test specification using the concept of validation function patterns, which include the test scenarios. This issue has been already mentioned in Chapter 3 and will be explained later in depth too. Afterwards, in *step III*, structures for test stimuli and concrete test signals are generated. This step occurs automatically with application of the transformations. The test control design can be added automatically too. In that case, *step IV* would be omitted. However, if the advantages of the test reactiveness are targeted, it should be refined manually. Finally, in the test execution and test evaluation phase in *step V*, the tests (i.e., test cases) may be executed and the test results obtained in the form of verdicts. At the same time the quality of the produced test system specification is also assessed.

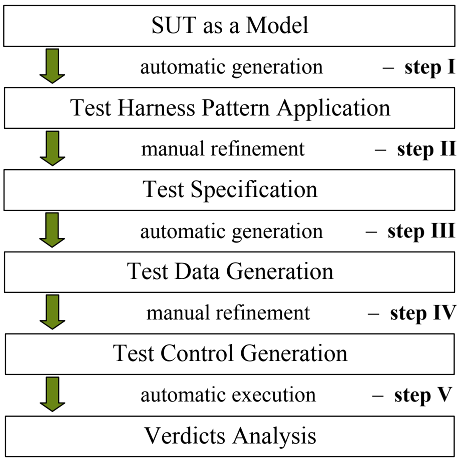

SUT as a Model
automatic generation – step I
Test Harness Pattern Application
manual refinement – step II
Test Specification
automatic generation – step III
Test Data Generation
manual refinement – step IV
Test Control Generation
automatic execution – step V
Verdicts Analysis

Figure 4.36: Test Development Process.

[30] An abstract test scenario means that it is valid for a certain generic description of the SUT behavior, without including the concrete data.

[31] In [ISTQB06] a test harness is defined as a test environment comprised of stubs and drivers needed to execute a test. In this thesis, the test harness pattern refers to the design level and is specified as an automatically created test frame including the generic hierarchical structure for the test specification. Together with the test execution engine (i.e, SL engine) it forms a test harness.

In Figure 4.37, a generic pattern of the test harness is presented. The test data (i.e., test signals produced for the test cases) are generated within the test data generator shown on the left-hand side. The test specification, on the right-hand side, is constructed by analyzing the SUT functionality requirements and deriving the test objectives from them. It includes the abstract test scenarios, test evaluation algorithms, test oracle, and an arbitration mechanism. The structural details and functionalities of the corresponding units will be elaborated in the next chapter[32].

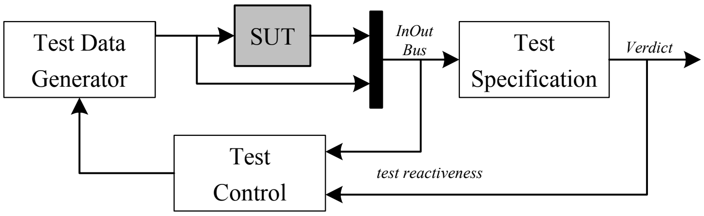
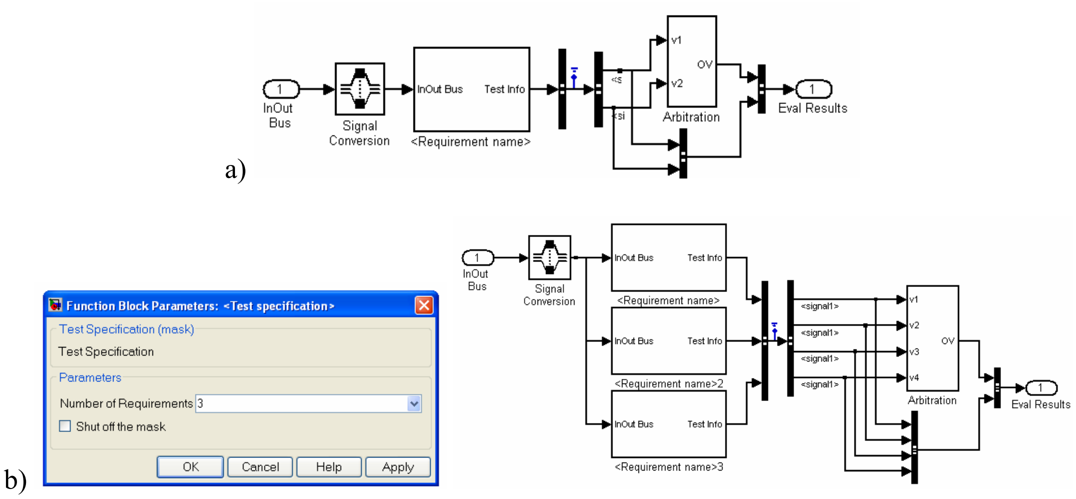


Figure 4.37: A Test Harness Pattern.

The test specification is built by applying the test patterns available in the MiLEST library. It is developed in such a way that it includes the design of a test evaluation as well – opposite to a common practice in the automotive domain, where the test evaluation design is considered last. Afterwards, based on the already constructed test model, the test data generators are retrieved. These are embedded in a dedicated test data structure and are derived from the test design automatically. The generation of test signals variants, their management, and their combination within a test case is also supported, analogous to the synchronization of the obtained test stimuli. Finally, the SUT model stimulated with the previously created test data is executed and the evaluation unit supplies verdicts on the fly.

The first step in the test development process is to identify the test objectives based on the SUT requirements. For that purpose a high-level pattern within the test specification unit is applied. The number of test requirements can be chosen in the graphical user interface (GUI) that updates the design and adjusts the structural changes of the test model (e.g., it adjusts the number of inputs in the arbitration unit). The situation is illustrated in Figure 4.38.

[32] Parts of the test process have been published in [ZSM07b], the test evaluation in [ZSM06, MP07], the test signals generation in [ZMS07a, ZXS08], whereas the test control and reactive testing in [Zan07]. This thesis is the most up to date and presents the most current progress.



Function Block Parameters: <Test specification>

Test Specification (mask)

Test Specification

Parameters

Number of Requirements 3

☐ Shut off the mask

Figure 4.38: A Pattern for the Test Requirement Specification.

a) Instantiation for One Test Requirement.

b) Instantiation for Three Test Requirements.

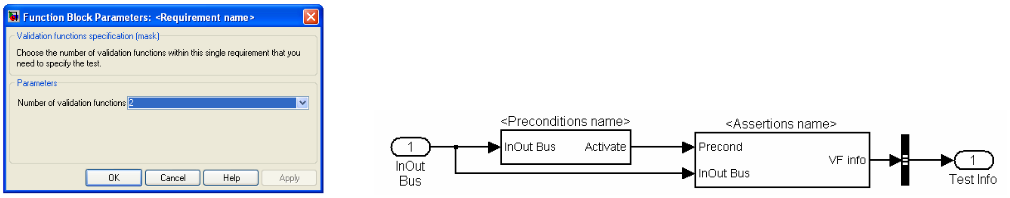
Next, validation functions (VFs) [ZSM06, MP07] are introduced to define the test scenarios, test evaluation, and test oracle in a systematic way. VFs serve to evaluate the execution status of a test case by assessing the SUT observations and/or additional characteristics/parameters of the SUT. A VF is created for any single requirement according to the conditional rules:

$$IF \quad preconditions \ \ set \quad THEN \quad assertions \ \ set \tag{4.8}$$

A single informal requirement may imply multiple VFs. If this is the case, the arbitration algorithm accumulates the results of the combined IF-THEN rules and delivers an aggregate verdict. Predefined *verdict* values are *pass*, *fail*, *none*, and *error*. Retrieval of the local verdicts for a single VF is also possible.

A preconditions set consists of at least one extractor for signal feature or temporally and logically related signal features, a comparator for every single extractor, and one unit for preconditions synchronization (PS).
An assertions set is similar, it includes, however, at least one unit for preconditions and assertions synchronization (PAS), instead of a PS.

VFs are able to continuously update the verdicts for a test scenario already during test execution. They are defined to be independent of the currently applied test data. Thereby, they can set the verdict for all possible test data vectors and activate themselves (i.e., their assertions) only if the predefined conditions are fulfilled.
An abstract pattern for a VF (shown in Figure 4.39) consists of a preconditions block that activates the assertions block, where the comparison of actual and expected signal values occurs. The activation and by that, the actual evaluation proceeds only if the preconditions are fulfilled.



Function Block Parameters: <Requirement name>

Validation functions specification (mask)

Choose the number of validation functions within this single requirement that you need to specify the test.

Parameters

Number of validation functions 2

Figure 4.39: Structure of a VF – a Pattern and its GUI.

The easiest assertion blocks checking time-independent features are built following the schema presented in Figure 4.40.

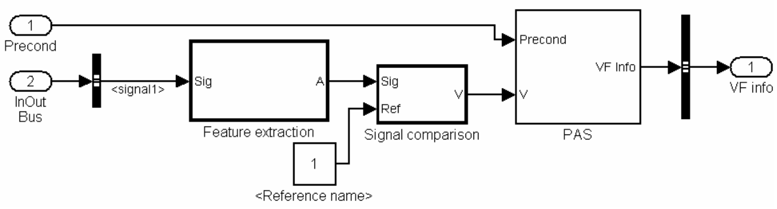


Figure 4.40: Assertion Block – a Pattern.

They include a SigF extraction part, a block comparing the actual values with the expected ones and a PAS synchronizer. Optionally, some signal deviations within a permitted tolerance range are allowed. Further schemas of preconditions and assertions blocks for triggered features are discussed in [MP07] in detail.

A further step in the test development process is the derivation of the corresponding structures for test data sets and the concretization of the signal variants. The entire step related to test data generation is completely automatic by merit of the application of transformations. Similar to the test specification side, the test requirements level for the test data is generated. This is possible because of the knowledge gained from the previous phase. The pattern applied in this step is shown in Figure 4.41.

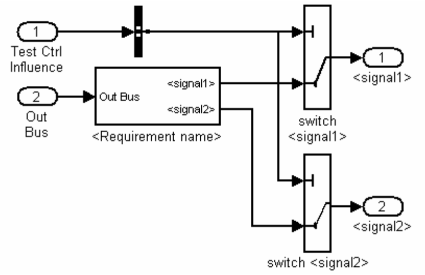


Figure 4.41: Test Requirement Level within the Test Data Unit – a Pattern.

Moreover, concrete SigFs on predefined signals are produced afterwards. The test signals are generated following a conditional rule in the form (see 4.9):

$$IF \quad preconditions \quad set \quad THEN \quad generations \quad set \tag{4.9}$$

Knowing the SigFs appearing in the preconditions of a VF, the test data can be constructed from them. The preconditions typically depend on the SUT inputs; however, they may also be related to the SUT outputs at some points in time. Every time a *SigF extractor* is present for the assertion activation, a corresponding *SigF generator* may be applied for the test data creation. Giving a very simple example: for detection of a given *signal value* in a precondition of a VF, a signal crossing this value during a default time is required. Apart from the feature generation, the SUT output signals may be checked for some constraints if necessary (cf. Figure 4.42). The feature generation is activated by a Stateflow (SF) diagram sequencing the features in time according to the default temporal constraints (i.e., *after(time1)*). A switch is needed for each SUT input to handle the dependencies between generated signals. *Initialization & Stabilization* block enables to reset the obtained signal so that there are no influences of one test case on another.

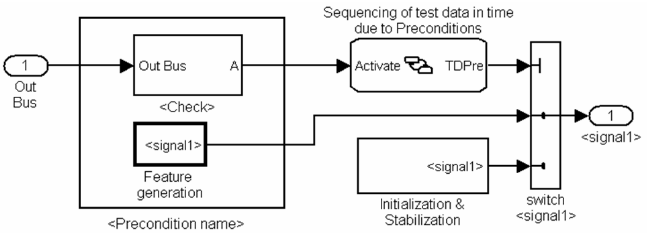


Figure 4.42: Structure of the Test Data Set – a Pattern.

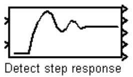
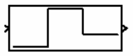
The patterns in Figure 4.42 and the concrete *feature generators* are obtained as a result of the automatic transformations. The general principle of the transformation is that if a given *feature or feature dependency extraction* is detected in the source (i.e., preconditions part of a VF), then the action to generate the target (i.e., *feature generator* in the test data structure) is performed. A set of transformation rules has been implemented. Afterwards, the concrete test data variants are constructed based on the generators obtained from the transformations. The assumption and necessary condition for applying the variants generation method is the definition of the *signal ranges* and *partition points* on all the stimuli signals according to the requirements or engineer's experience. Equivalence partitioning and boundary value analysis are used in different combinations to then produce concrete variants for the stimuli.

When a test involves multiple signals, the combination of different signals and their variants have to be computed. Several combination strategies are known to construct the test cases – *minimal combination, one factor at a time*, and *n-wise combination* [LBE$^+$04, GOA05]. Combination strategies are the selection methods where test cases are identified by combining values of different test data parameters according to some predefined criteria. In this thesis, the first two strategies have been used.

A similar sequencing algorithm like for the test data applies for ordering the test cases on a higher hierarchy level while dealing with a number of requirements. This aspect is called test control. A traditional understanding of the control makes it responsible for the order of test cases over time [ETSI07]. It may invoke, change, or stop the test case execution due to the influence of the verdict values coming from the test evaluation. Thus, the test cases are sequenced according to the previously specified criteria (e.g., *pass* verdict). An extended definition of the test control is considered in Section 5.5.

The test patterns used for realizing a test specification proposed in this thesis are collected in a library and can be applied during the test design phase. In Table 4.4 two patterns are presented, each of them corresponding to a selected testing activity. Since they will be applied in the case studies part in Section 6.4, their meaning will be explained afterwards. The full collection of MiLEST patterns enabling the entire test system to be built is attached to this thesis in Appendix B.

Table 4.4: Illustration of Reasoning about Patterns.

Test Activity	***Test Pattern Name***	***Context***	***Problem***	***Solution Instance***
Test evaluation	Detect SigF	Test of a control unit	Assessment of a control unit behavior in terms of a selected SigF	 Detect step response characteristics
Test data generation	Generate SigF	Evaluation of a step response function	Generation of the proper signal to stimulate a selected feature on the SUT output signal	 Generate complete step

After the test specification has been completed, the resulting test design can be executed in SL. Additionally, a report is generated including the applied test data, their variants, test cases, test results, and the calculated quality metrics.

4.5 Related Work

4.5.1 Property of a Signal

Property of a signal, called SigF in this thesis, has been introduced in hybrid Sequence Charts notation [GKS99] used for describing the behavior of hybrid systems. The language is based on Message Sequence Charts [ITU96, ITU99], including some concepts of timing diagrams [ABH+97]. Already there, the signal has been partitioned according to its characteristics and constraints put on it. This technique will be recalled in Chapter 6 to illustrate some of the developed concepts.

[GW07] indicates that the descriptive approach to the signals reveals some advantages over the commonly-used constructive approach, especially in the context of SUT behavior evaluation. Here, an obvious link to the test evaluation in MiLEST exists.

Further on, the consortium developing the Testing and Test Control Notation (TTCN-3) for embedded systems [Tem08] incorporates the paradigm of SigF into the ongoing research on the test assessment functions.

Finally, the timing relations between signals classified in [GHS+07] contribute to the development of temporal expressions within the test specification proposed in MiLEST.

4.5.2 Test Patterns

Referring to the *test patterns*, [Bin99] describes *object-oriented test strategy patterns*. They handle several strategies to derive test cases for testing object-oriented software. However, this definition cannot be adapted in this thesis as object orientation is out of its scope. [Din08] elaborates a set of methods and patterns to design and implement efficient performance tests. These are also only related to the test patterns presented in this thesis. In [Neu04] Real-Time

Communication patterns are used in the form of time relations among communication operations. They describe real-time requirements related to delay, throughput, periodic events, and jitter. These patterns are applicable to the telecommunication systems mainly.

[Neu04] also presents a detailed survey through test patterns regarding a number of criteria. According to his classification the proposed patterns are categorized as *functional* in the context of the *test goals*, as *test design* patterns in the sense of the *test development* and as *component level* considering the *scope of testing*.

[TYZ$^+$03] argue that developers often specify embedded systems using scenarios and a typical medium-size system has hundreds of thousands of scenarios. Each scenario requires at least one test case, which in turn requires individual development and debugging. *Verification patterns* (VP) are proposed to address this problem. The VP approach classifies system scenarios into patterns. For each *scenario pattern* (SP), the test engineer can develop a template to test all the scenarios that belong to the same pattern. This means that the engineer can reuse the test templates to verify a large number of system scenarios with minimum incremental cost and effort. Each scenario has preconditions (causes), postconditions (effects), and optional timing constraints. A SP [TYZ$^+$03] is defined as a specific temporal template or cause-and-effect relation representing a collection of requirements with similar structures. A VP [TYZ$^+$03] is a predefined mechanism that can verify a group of behavioral requirements that describe similar scenarios. A GUI-based specification tool to facilitate the scenario specification is available. In this work the focus is put on the test specification patterns, which mainly relate to the test behavior similarly to [TYZ$^+$03]. However, both discrete and continuous signals are handled, while [TYZ$^+$03] addresses only scenarios describing discrete behavior.

4.6 Summary

In this chapter the second set of the research questions given in the introduction to this thesis has been addressed. In particular, a new way for handling the *discrete and continuous signals* at the same time, based on the SigFs, has been provided. By that, a first sketch of the *test framework* realizing this concept has been given, indicating the design decisions in terms of test generation and test evaluation.

The main technical intention of this chapter was to introduce a new way of signal description by application of the SigFs. These have been discussed in Section 4.1. Further on, based on the previous assumptions, Section 4.2 introduced the classified means for signal generation and signal detection. The challenges and limitations of the realization have been discussed.
In Section 4.3, test patterns have been investigated, with a particular emphasis on the hierarchical architecture of the proposed test system. The MiLEST test patterns have been attached to this thesis as a table in Appendix B. These will be carefully reviewed in Chapter 5 and applied to a number of case studies in Chapter 6.
Furthermore, Section 4.4 elaborated on the proposed test specification process and its development phases starting from requirements analysis until the test execution. Finally, the related work on SigFs and test patterns has been described in Section 4.5.
The detailed discussion on the test development process artifacts will be continued in Chapter 5.

5 The Test System

"I dream, I test my dreams against my beliefs, I dare to take risks,
and I execute my vision to make those dreams come true."

- Walt Disney

The upcoming chapter is related to the previous one since the considerations on the new test paradigm introduced there, are continued here. In particular, *a test framework* is provided in order to *automate* the creation of concrete *test systems*. This is possible by application of *test patterns* that are organized into a hierarchy on different abstraction levels.

The fundamental approach to the signal features (SigFs), their classification, generation, and detection mechanisms enable to synthesize the entire architecture of the test system. This is done in Section 5.1 in order to exploit the SigF for testing purpose. Different abstraction levels for both test specification (TSpec) and test data generation (TDGen) are provided. The full consolidation of the architecture levels can be additionally found in Appendix C. Their realization is possible applying the Model-in-the-Loop for Embedded System Test (MiLEST) method proposed in this thesis. In the TSpec, also the test evaluation is modeled as Section 5.2 emphasizes. The advantages of designing the architecture as such are revealed especially at the validation function (VF), feature generation, and feature detection levels, where the real test stimuli creation and test assessment of the SUT behavior takes place. Furthermore, the link to the TDGen is established by application of the automatic transformations, which in consequence, supports the automatic generation of the test signals. The details of these activities are given in Section 5.3. The obtained test patterns are classified and categorized. In contrast to the related work, the numerous patterns proposed herewith are very granular and represent different abstraction levels while specifying the tests. This gives the possibility to navigate through the test system easily and understand its contents from several viewpoints immediately.

In Section 5.4, a description of signal variants generation is discussed, combination strategies for test case construction are reviewed and variants sequencing at different levels is considered. This contributes to the concept of automatic and systematic test data generation representing a comprehensive advantage over the existing solutions.

Section 5.5 introduces the concept of reactive testing. There, also the test control is investigated. The concept of online test data manipulation is introduced that contributes to test time reduction at the end. Afterwards, Section 5.6 underlines the progress on integration level testing. In Section 5.7, the execution of the resulting test model is considered. Also, the importance of the test report is shortly mentioned.

Finally, Section 5.8 gives insights into the related work in the context of the test evaluation process based on the SigF, other transformation approaches and the ongoing work of the author of this thesis towards some extensions affected by the paradigm presented here. The summary in Section 5.9 completes this chapter.

5.1 Hierarchical Architecture of the Test System

The test development process described in the previous chapter together with the abstract architecture of the test system enable to apply the concepts of SigF generation and extraction mechanisms while building test models systematically. In this context, the term architecture of the test system is understood as a hierarchically structured test model. Consequently, the formal test specifications in SL/SF reflect the structure of the system requirements. The primary goal behind the architecture is a hierarchical structure that serves for gaining abstraction.
Since the aim of the test system is not to provide the means for testing single signal properties but for validating complete SL/SF SUT models, independently of their complexity, structuring the test models in a proper way contributes to the scalability and reusability of the solution. Moreover, traceability of the test development artifacts and transformation potentials can be identified.

The structure of the test system consists of four different levels (see Figure 5.1) that can be built systematically. This makes the test system less error-prone while also leaving the test engineers plenty of scope for developing the complete test specification. Figure attached to this thesis in Appendix C provides a deeper insight into the architecture. Even though the diagram presented there has a general character, a similar format to the signal flow diagram notation of SL/SF has been used. Also, the scheme is used to describe the different level details in the following paragraphs.

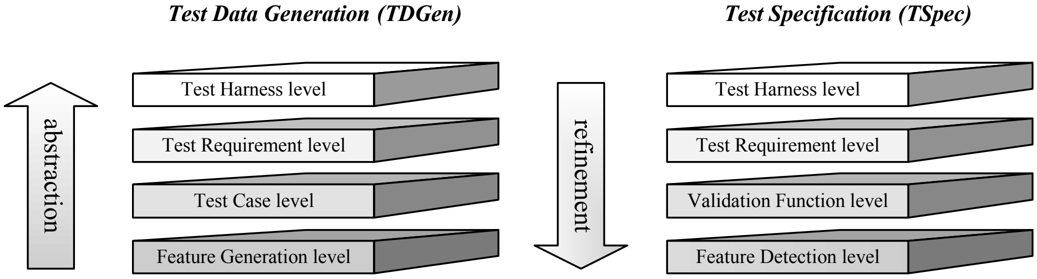


Figure 5.1: Hierarchical Architecture of the Test System.

At the highest abstraction level the test harness appears. It is the same for both test data generation (TDGen) and test specification (TSpec) lanes. Here, apart from the SUT model, TDGen, TSpec, and test control units exist. Also, the connections between the corresponding units are covered. All SUT input and output signals are collected in advance and provided as a single bus signal to the TSpec. TSpec includes the behavioral test scenarios that are eventually evaluated. Due to this functionality, the TSpec can sometimes be called test evaluation. Additionally, any other signals that might be of interest for the test can also be fed into the SUT. In order to provide a full TSpec and make the test evaluation independent of the current SUT input signals all SUT interfaces should be covered. The test evaluation block outputs the overall test verdict.

One level below, the test requirement level divides the TSpec into the different SUT requirements to be tested. At this level, on the one hand, the test data sets are collected according to the requirements they are assigned for. On the other hand, the test evaluation mechanisms are separated into different blocks according to the requirements they check. Each requirement block outputs all relevant test information of that requirement. An arbitration algorithm extracts the overall verdict from the structured test results. Single requirement verdicts can also be obtained.

The third level is a bit more complicated. Considering the TDGen scope, the test data sets forming the test steps are stored and their sequencing into test cases is supported. Test steps represent a combination of features designed in the preconditions of a VF, appropriately.

Within the TSpec scope, the corresponding VFs containing preconditions-assertions pairs appear. Here, the abstract behavioral test scenarios are included, with an emphasis on properties and their interdependencies to be asserted.

Finally, at the lowest abstraction level, SigFs are managed, generated, and extracted, respectively as described in Section 4.2.

5.1.1 Test Harness Level

The test harness level consists of the SUT model usually designed in SL/SF, TDGen unit, TSpec unit, and test control as already mentioned in Section 4.4 and shown in Figure 4.37. All SUT inputs and outputs or any other relevant signals required by the actual VFs (e.g., intermediate) are collected using *Bus Creator* blocks and passed to the SUT as a single bus, called *InOut Bus*. A good practice is to name the signals included in the *InOut Bus* so as to let the test evaluation unit to extract the correct signals for a particular VF. This activity is done automatically by application of transformations described in Section 5.3.

Additionally, the input and output signals are split forming two separate abstract buses, application of which will be explained analyzing the concrete implementation in Chapter 6.

Furthermore, at this level the test configuration is established. Apart from the default elementary units, the test components, or plant model may be comprised.

5.1.2 Test Requirement Level

The test requirement level for TDGen (see Figure 5.2) is built using a single subsystem for each tested requirement and a *Selection* block being a multi-port switch for switching between different signals. The number of *Selection* blocks is equal to the number of test stimuli that are to be passed on to the SUT. These SUT inputs flow out of every requirement to the *Selection* blocks. The signals are ordered by matching their names with the corresponding *Selection* blocks. The maximum number of signals generated within a requirement is equal to the number of all SUT inputs. Thereby, the size of the *Selection* block is determined by this number too. Nevertheless, not all of them must be generated within one requirement since not all of them play a significant role for a particular test objective. In the case when an input is not included in the requirement subsystem, a default signal is produced. These and other rules will be described in Section 5.3 in detail.

Additionally, at the test requirements level, the *Test Control* signal is forwarded to all the *Selection* blocks. Its primary task is to inform the *Selection* in which order the scenarios should (here test cases) be generated and how they relate to each other. This issue will be explained in Section 5.5 in depth.

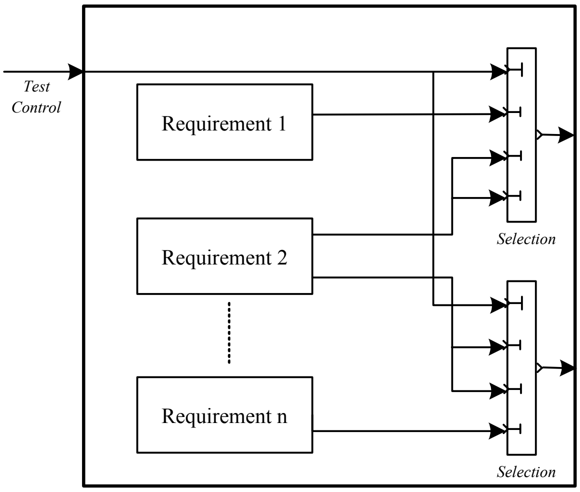


Figure 5.2: Fundamental Structure of the Test Requirement Level – TDGen View.

The test requirement level is actually a pure abstraction level. From the TSpec point of view (see Figure 5.3), each requirement consists of several VFs. A mask of this block enabling to set the number of VFs has already been shown in Figure 4.39.

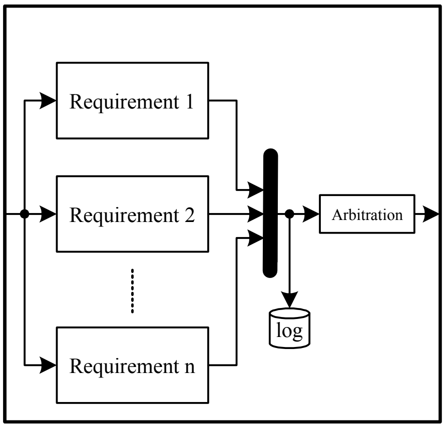


Figure 5.3: Fundamental Structure of the Test Requirement Level – TSpec View.

At this point a clear distinction between the TSpec and test evaluation is done. Starting from here not only are test scenarios considered, but also their evaluation, in terms of assertions check, is taken into account. By that, the TSpec becomes to be the test evaluation at the same time. The specified test scenarios within each requirement are assessed resulting in a verdict. The evaluation issues will be described deeper at the further levels, representing more concrete test solutions.

At the test requirement level, the *InOut Bus* coming from the test harness level is passed to each requirement subsystem, which in turn, lets the test system to evaluate the SUT and collect the assessment information, including verdicts and arbitration mechanism.
The test results of all requirements are logged using the signal logging capabilities of SL. Technically, the pin after the *Bus Creator* block in Figure 4.38 indicates that the signal values are logged to the MATLAB (ML) workspace during the simulation. The logged signals include all information about the test assessment. Hence, the log file size depends on the size of the entire test evaluation system. All abstraction levels can be identified in the logged structure.

Figure 5.4 additionally shows the implementation of an arbitration algorithm. It determines the global overall verdict from all local verdicts of the VFs. As a matter of example, three local verdicts are passed on. The minimum among the local verdicts constitutes the global, i.e., overall one.

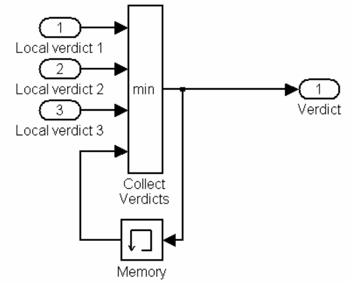


Figure 5.4: Arbitration Mechanism.

The arbitration algorithm is based on the minimum computation, because the verdicts are defined in descending priority (cf. the last paragraph of Section 5.1.4). The feedback *Memory* block enables the algorithm to retain the worst verdict found to date. Thus, the overall verdict equals to pass if any local verdict has been at least once pass and no local verdict has been fail or error during the test. An important implementation detail is that the initial output value of the *Memory* block must be set on a value not less than two (in contrast to the standard value that is zero). Otherwise one for pass or two for none can never be reached. The value of the overall verdict once set on a lower case, will it never come back on the upper case.

Additionally, from the technical perspective, the data type conversion for the *InOut Bus* is supported at this level (cf. Figure 4.38). *Signal Conversion* block is applied for automatically casting the data types of all signals of a bus to the format required by the destination blocks.

The requirement blocks in the test requirement level, for both TDGen and TSpec views should be named, which improves the readability and enables a straightforward tracing to the textual requirements. This option is technically supported by several interfaces between requirements tools and SL/SF[33].

[33] The requirements tracing [SLVV] is possible using the Requirements Management Interface for Telelogic DOORS® software [TelD] or selection-based linking for Microsoft® Word and Excel® documents – to name the two examples.

5.1.3 Test Case Level – Validation Function Level

The test case level is a level related entirely to the generation of test cases – on the TDGen lane. It consists of *Test Data* subsystems, a *Generation Sequence* and *Selection* blocks (see Figure 5.5). The *Test Data* blocks correspond to the *Preconditions* subsystems that are to be found on the same abstraction level, called validation function level, but on the TSpec side. One *Test Data* subsystem contributes to the concept of the so-called test step. The test case is composed of a sequence of such test steps that are ordered in *Generation Sequence* for one single requirement. This means that the test case is constructed by the *Generation Sequence* block and *Test Data* blocks. The maximum number of *Selection* blocks (i.e., concrete multi-port switches) is determined by the number of SUT inputs.

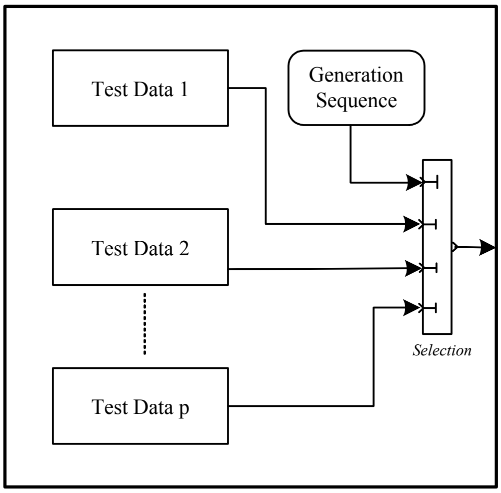


Figure 5.5: Fundamental Structure of the Test Case Level.

At this level, similarly as at the test requirement level, not all the SUT inputs must be constrained. The inputs, for which no SigFs have been explicitly defined, obtain the default values. Such a case is shown in an exemplified implementation in Figure 5.6, which will be explained in Section 6.2 in detail. Here, the phi_Brake signal has not been constrained.

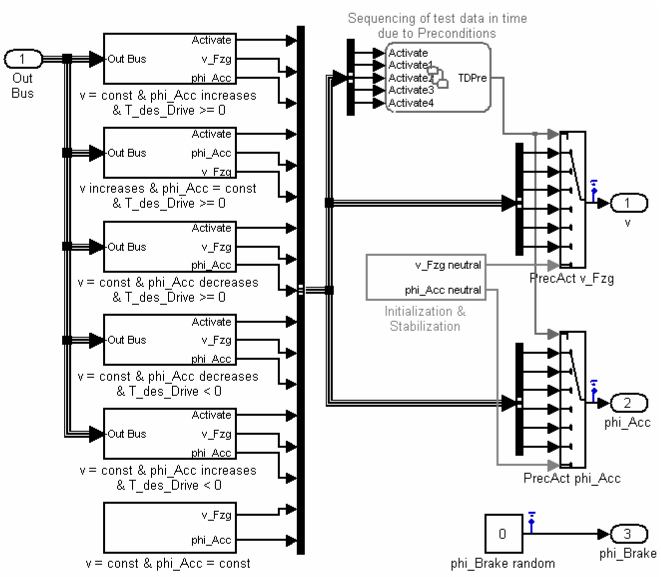

1
Out Bus
Out Bus
Activate
v_Fzg
phi_Acc
v = const & phi_Acc increases & T_des_Drive >= 0
Out Bus
Activate
phi_Acc
v_Fzg
v increases & phi_Acc = const & T_des_Drive >= 0
Out Bus
Activate
v_Fzg
phi_Acc
v = const & phi_Acc decreases & T_des_Drive >= 0
Out Bus
Activate
v_Fzg
phi_Acc
v = const & phi_Acc decreases & T_des_Drive < 0
Out Bus
Activate
v_Fzg
phi_Acc
v = const & phi_Acc increases & T_des_Drive < 0
v_Fzg
phi_Acc
v = const & phi_Acc = const
Sequencing of test data in time due to Preconditions
Activate
Activate1
Activate2
Activate3
Activate4
TDPre
v_Fzg neutral
phi_Acc neutral
Initialization & Stabilization
PrecAct v_Fzg
1
v
PrecAct phi_Acc
2
phi_Acc
0
phi_Brake random
3
phi_Brake

Figure 5.6: Exemplified Structure of the Test Case Level.

Thus, the *real* number of Selection blocks is equal to the number of the SUT inputs constrained in the preconditions. Further on, the size of a single Selection block is usually the same as the number of *Test Data* subsystems. It is assumed that every *Test Data* set includes different types of SigF generators for a given SUT input. The signals coming out of the *Test Data* sets are ordered by matching their names with the corresponding Selection blocks in the sequence of their appearance.

Additionally, an *Initialization/Stabilization* block (cf. middle of Figure 5.6) may exist for resetting the signals between different test steps so that the test behavior of the first one does not influence the next one. Since this situation is a specific one, it will be used for explanation of the concrete case studies in the upcoming chapters.

Finally, an *Out Bus* can be required (cf. left side of Figure 5.6). It enables the values of the SUT output, among other signals, to be checked. If predefined conditions are fulfilled, a given test step starts to be executed; otherwise another one, unconstrained is chosen. This concept contributes to the test reactiveness. The *Generation Sequence* block serves as a unit for ordering the test steps in time. If any dependencies of the TDGen on the SUT outputs exist, they are handled within this element.

As already described in Section 4.4, VFs are the implementation of IF-THEN rules. The VF is formed by preconditions-assertions block which is reflected in the validation function level. Herewith, the independence of the applied test signals during the test execution is obtained on the one hand. On the other hand, the test evaluation system checks the specified test scenario constantly and simultaneously, not just at certain time steps determined by a test case. At this point the discussion on the relation between the TSpec and test evaluation from the previous section can be recalled. The test evaluation system represents a formal and systematic TSpec, indeed. The same applies vice versa in this case. Moreover, the verdicts set for the different assertions do not directly link to a test case. A verdict primarily belongs to its corresponding VF and therewith to a requirement as well.

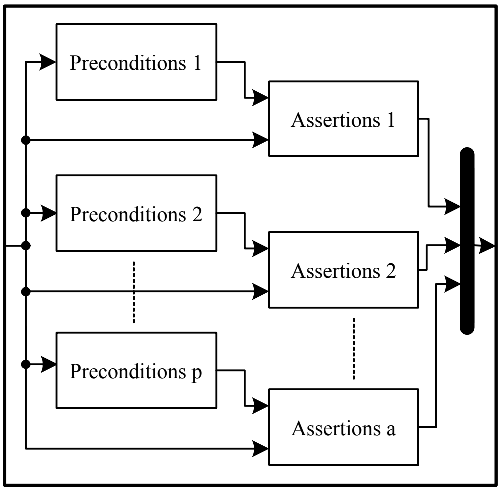


Figure 5.7: Fundamental Structure of the Validation Function Level.

In Figure 5.7, the fundamental structure of the validation function level is shown. Groups of preconditions and assertions blocks can be recognized, all preconditions having the same input – the signal bus containing all SUT relevant signals. The *Precond* signal bus is connecting preconditions and assertions. A *Bus Creator* block collects the assertions outputs, which enables them to be linked with the corresponding VFs.

Similarly as for the upper level, a good practice within this level is to give the different blocks the names of the activities performed inside. With this practice, readability and quick understandability is supported. The elements may be traced to the requirements easily too.

5.1.4 Feature Generation Level – Feature Detection Level

The feature generation level is the implementation of the generation algorithms for the SigFs. Since they have been already introduced in Section 4.2, the explanation of this level falls short. Every SigF is embedded in a *Feature Generator* subsystem. The feature generation works according to the generic algorithm, whereat numerous variants of the SigF are constructed. These are produced as a result of transformations described in Section 5.3. The created signals are passed on to the test case level and managed there further on. Additionally, a log file is produced for each signal (cf. Figure 5.8): it is applied for generation of the test report and realization of the quality metrics described in Chapter 7.

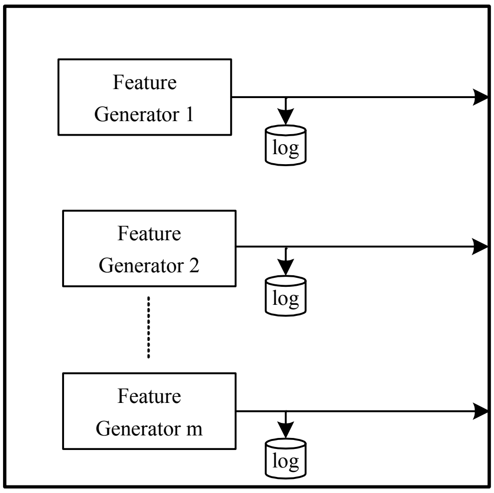


Figure 5.8: Fundamental Structure of the Feature Generation Level.

The structure handling the variants management is shown in Figure 5.9. The insights of the *Feature Generator* subsystem are shown to illustrate the generation of feature representatives. In this example, two variants are produced.
The activation of variants on the test data level is synchronized with the test control. It is assured due to the application of the *From* block (cf. Figure 5.9) retrieving the variant number from the *Goto* block that is specified in the test control (cf. Figure 5.26 – in Section 5.5). Further details on variants management will be described in Section 5.4.

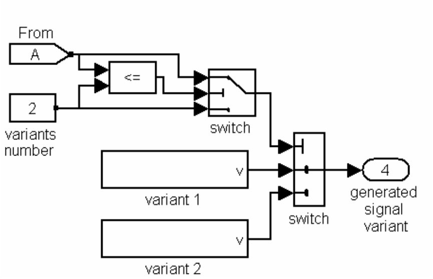


Figure 5.9: Variants Management Structure.

The feature detection on the same abstraction level, but from the TSpec perspective, is the technical realization of signal evaluation. At this level more signal evaluation units appear and relate to each other by a logical AND operator. Each atomic signal evaluation unit consists of a feature extraction block in conjunction with a signal comparison block and the value of a reference SigF (see Figure 5.11 and Figure 5.14).

Following the IF-THEN rule that propagates the SigFs in preconditions-assertions pairs, their synchronization is required. Two cases must be distinguished since precondition blocks produce a common activation signal set for the assertions, while the assertions deliver a set of verdicts and related information. Consequently, both cases should be realized separately in *PS* and *PAS* blocks as mentioned in Section 4.4. Thus, the upcoming discussion will be split into two topics: the realization of preconditions and assertions. Before, a short description of the synchronization algorithm assumption will be given. The granular implementation details concerning the mechanism for both elementary parts of the feature detection level can be found in [MP07] (pages: 55 – 66 and 71 – 83, respectively).

Detection of TI features can be modeled as an identification of TDD features that are triggered at every time step. Respectively, detection of TDD features can be described as the extraction of TID features that have a constant delay as shown in Figure 5.10. This principle is utilized in the implementation of the synchronization algorithm. Firstly, the detection mechanisms of all features are transformed to the most complex form including A, T, and R signals. Then, a common activation signal bus is generated; it is built from the feature, trigger, and reset signals and a mode signal indicating what kind of feature description has been applied – TI, TDD, or TID. The assertions are only activated at these time steps when all preconditions are active [MP07].

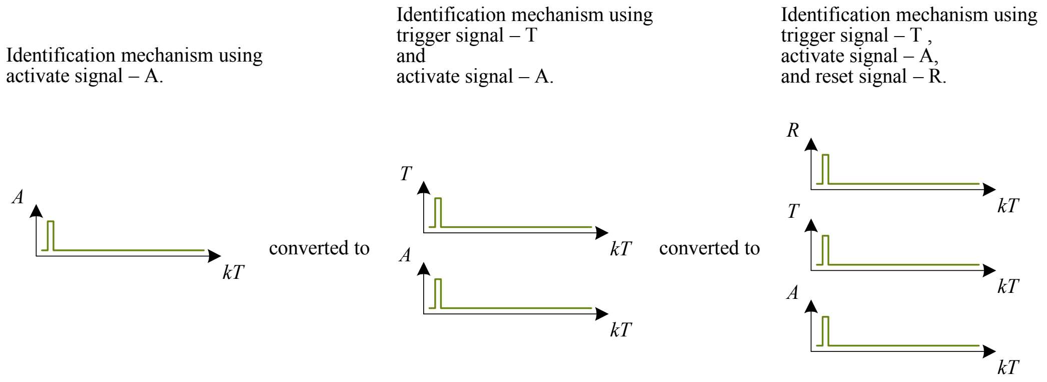


Figure 5.10: Conversions of the identification mechanisms for TI – TDD and TDD – TID features.

Test Specification: Feature Detection Level: Preconditions. The feature detection level for the preconditions is structured as shown in Figure 5.11. In the schematic structure only TI features are considered, since the extraction blocks only output a *feature signal*. In the implementation, however, the *trigger* (T), and *reset signals* (R) can be utilized by the PS block for synchronization too.

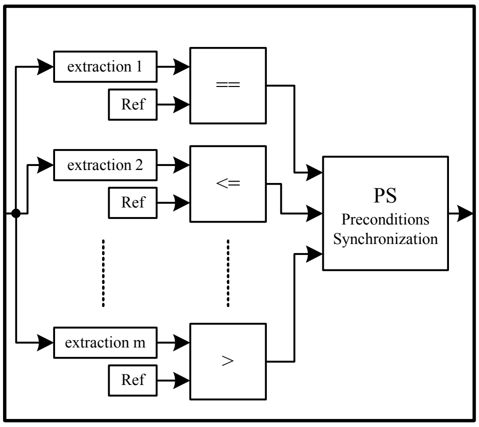
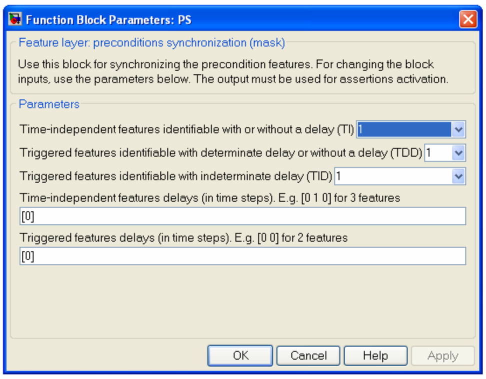


Figure 5.11: Fundamental Structure of the Feature Detection Level for Preconditions.

The comparison block[34] supports basic comparison operations, namely ==, ~=, ≤, ≥, <, and >, but also more-flexible comparison forms that include tolerance ranges. The comparison blocks transform the *feature signal* to a boolean value, outputting true for a successful check, i.e., when the feature under extraction exist. The resulting signal is denoted as *activation signal* (A), next to T and R.

Thus, the preconditions synchronization block has as many A inputs as features are extracted, as many T inputs as TDD and TID features are available, and as many R inputs as TID features are checked. The number of TI, TDD, and TID features can be set in the PS block mask shown in Figure 5.12.


Function Block Parameters: PS

Feature layer: preconditions synchronization (mask)

Use this block for synchronizing the precondition features. For changing the block inputs, use the parameters below. The output must be used for assertions activation.

Parameters

Time-independent features identifiable with or without a delay (TI) 1

Triggered features identifiable with determinate delay or without a delay (TDD) 1

Triggered features identifiable with indeterminate delay (TID) 1

Time-independent features delays (in time steps). E.g. [0 1 0] for 3 features

[0]

Triggered features delays (in time steps). E.g. [0 0] for 2 features

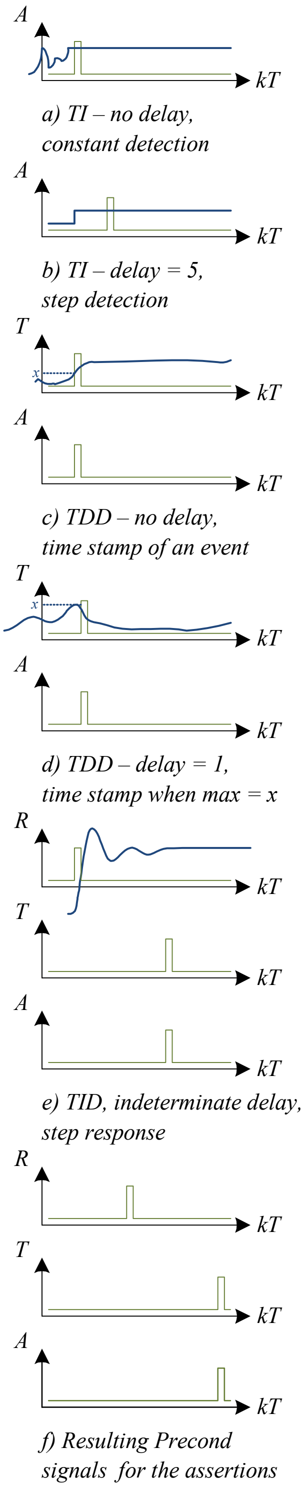
[0]

Figure 5.12: Preconditions Synchronization Parameter Mask.

[34] Simulink® Validation and Verification™ library offers a set of comparison blocks. Although SL includes the possibility of disabling them after the system has been validated successfully, they could be used here as well.

When TI or TDD features are extracted, their delays have to be specified using a vector notation. They need to be provided in simulation steps, i.e., *x* seconds delay would be introduced in the form of x^* *time step size* delay.

The basic functionality that must be covered by the PS block will be illustrated using the example presented in Figure 5.13, introduced initially by [MP07]. Therein, the extracted signals of five different features are shown (Figure 5.13, cases a to e). The first feature represents constant detection, the second one – step detection with a 5 units delay. The next figure shows the time when a signal crosses a value x (i.e., extraction of the time stamp of an event). The fourth feature identifies the local maximum of value x with a unit delay. The last feature measures some step response characteristic. Given this concrete situation, the task of the PS block is to deliver the activation information to the assertion block (cf. Figure 5.13, case f).



a) TI – no delay, constant detection

b) TI – delay = 5, step detection

c) TDD – no delay, time stamp of an event

d) TDD – delay = 1, time stamp when max = x

e) TID, indeterminate delay, step response

f) Resulting Precond signals for the assertions

Figure 5.13: Preconditions Synchronization – an Example.

Introduction of a delay presents the straightforward reasoning about A signal. Hence, taking the features in Figure 5.13, cases a and b as an example, the feature identifiable with a shorter delay must be fictitiously delayed by the delay difference, in order to filter the delay out. This time shift guarantees that all features are evaluated at the same time. If the A signal in Figure 5.13b is delayed by 5 time steps, both A signals will be triggered at the same time step. In consequence, the maximum delay within the preconditions determines the size of the preconditions delay. It is calculated using the formula:

$$\Delta\tau_i = \tau_{max} - \tau_i \tag{5.1}$$

$\tau_{max} = max(\tau_i)$ represents the maximum delay among all the SigFs present in the preconditions.

Thus, the signals in Figure 5.13c need to be delayed by 5 time steps, whereas the signals in Figure 5.13d needs to be additionally delayed by 4 time steps.
If a *trigger signal* is extracted, the preconditions become triggered automatically. Whenever the triggered feature is not available, the preconditions are not available either, since they are all related by the AND-operator. In that case, all TI features must be transformed to TDD features, so that they obtain a fictitious T signal as well. Its value is the same as the value of A signal. A TDD feature with identical T and A signal is equivalent to a TI signal.

Considering the TID features the situation is a bit more complex. In the case of having only one precondition, whenever the R signal becomes true, the preconditions are *potentially* active. Then, their *real* activation is confirmed only when T and A signals become active too. All three signals, R, T, and A must be passed to the assertions, so as to inform them about both the potential and real activation.
Combinations of TID features only activate the preconditions if all their R signals become active at the same time and the R signals then remain inactive until all T and A signals have become active. Each T-A pair must become active simultaneously, but the delays of the different T-A pairs do not have to be the same. If the T signal becomes active without the corresponding activation of its A partner, the preconditions do not become active. A new activation of R restarts the synchronization process. Again, the highest delay applies for all preconditions. Combinations of TID features with TI or TDD features are managed by converting the latter to TID. This is achieved by generating an R signal identical to the T signal of the TDD feature.
The R signal of TID features cannot be delayed. This limitation was introduced to reduce the complexity of the synchronization algorithm, meaning that the extraction mechanism for the R signal needs to be delay-free. When different feature types are combined, all three TID feature description signals R, T, and A are delayed by the maximum [MP07].

Test Specification: Feature Detection Level: Assertions. The assertions at the feature detection level have the same structure as the preconditions, i.e., SigFs are extracted and related conjunctively. Only when all preconditions are active, can the assertions be active.
However, if a large set of IF-THEN-rules is considered, it would be possible to find more than one rule with exactly the same preconditions, but with different assertions. If these rules belong to different requirements, the preconditions set will repeat in each rule. Such a case does not have to be designed this way, but it appears to be a good practice for at least two reasons. Firstly, the requirements may slightly change. As a result previously the same preconditions would be altered too, which would consequently lead to their separation anyway. Secondly,

every single assertion should deliver a separate verdict in order to localize a *fail* as efficient and effective as possible.

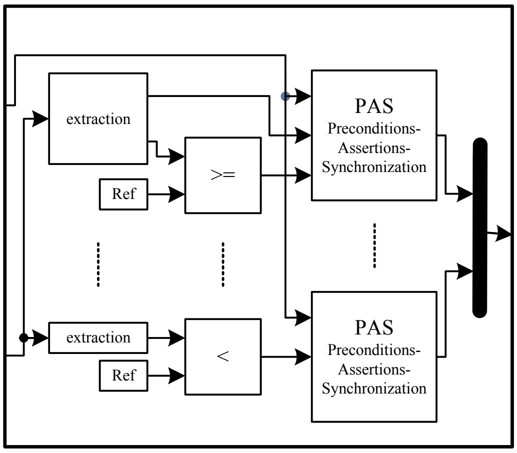


Figure 5.14: Fundamental Structure of the Feature Detection Level for Assertions.

In this work, every assertion is treated as an autonomous unit, delivering its own verdict. Regarding the implementation, the IF–THEN rule (4.8) is transformed to the set given in (5.2) without losing the previous semantic. Therefore, the way how the single assertions are related to each other can be left open.

$$\begin{array}{l} IF\ Precondition_1 \wedge Precondition_2 \wedge \ldots \wedge Precondition_m\ THEN\ Assertion_1 \\ IF\ Precondition_1 \wedge Precondition_2 \wedge \ldots \wedge Precondition_m\ THEN\ Assertion_2 \\ \ldots \\ IF\ Precondition_1 \wedge Precondition_2 \wedge \ldots \wedge Precondition_m\ THEN\ Assertion_n \end{array} \qquad (5.2)$$

Apart from the feature extraction, each assertion has to be synchronized with the preconditions. This task is performed by the preconditions-assertions synchronizer. Here, the activation signal A from preconditions activates the assertions and PAS functionality at the same time. Each PAS block consists of a PS block for exactly two features, complemented by algorithms capable of setting verdicts and delays independently of the feature types used. Every assertion delivers a verdict, a verdict delay, and further verdict-related information at every simulation time step. The information from the different assertions is collected and recorded separately in the requirements level, which assures a separation of concerns. PAS block outputs a signal bus with the evaluation information. All PAS buses are collected in order to output a single bus for all assertions at once.

The setting of different verdicts that occurs in PAS contrasts with the simple monitoring that PS block does.

Also, the signal comparison blocks slightly differ from the ones for the preconditions. The actual comparison mechanisms remain the same, but the blocks collect further test-related information. With this practice, fundamental elements for reconstructing and understanding the test verdicts offline may be retrieved. *Offline* means after the execution of a test, in contrast to *online* which means during its execution. Each test verdict can be related to a concrete asser-

tion, and in consequence, to the corresponding requirement and to a specific time step. Also, the test behavior can be reproduced offline using the test evaluation system.
The verdicts that can be applied are:

- none (2) – when no other verdict can be set or when the preconditions are not active
- pass (1) – when the SUT functional behavior is correct
- fail (0) – when the SUT behavior does not correspond to the expected one
- error (−1) – when the test system contains errors (e.g., the PAS input signals have an invalid format).

The arbitration mechanism has been described in Section 5.1.2 and illustrated in Figure 5.4. There is a default arbitration algorithm. It is used while delivering the verdict for a single assertion, a requirement evaluation, test case, or the entire test suite. Verdicts are ordered according to the rule: *none* < *pass* < *fail* < *error* following the standard [UTP].

5.2 Test Specification

As introduced in Section 4.4 and explained in Section 5.1.3, a VF is the fundamental part constituting the TSpec. Referring to the rule (4.8), the preconditions set and assertions set are propositional variables that stand for any propositions in a given language. The preconditions set is called the *antecedent* and the assertions set is called the *consequent*, while the statement as a whole is called either the *conditional rule* or the *consequence*. Assuming that the conditional statement is true then the truth of the antecedent (i.e., fulfilling the constraints in the preconditions set) is a *sufficient condition* for the truth of the consequent (i.e., activation of the assertions set), while the truth of the consequent is a *necessary condition* for the truth of the antecedent [Men97].

The conditional rule given in (4.8) is understood as an abstract test scenario describing a set of preconditions that must be fulfilled so as to assess the expected behavior of the SUT (i.e., activate the assertions). Concretization of such a scenario at the VF level is also possible, but usually it is more effective to design more generic test specifications.
The main difficulty that the test engineer needs to overcome at this place is to create reasonable IF-THEN statements from the SUT requirements so as to design flexible, high-quality tests. Thus, some test modeling guidelines are provided in the following in order to clarify the process of TSpec.

Generally, the rule holds that the preconditions should be determined by the constraints set on the SUT input signals, whereas the assertions should check the SUT outputs. This is the natural manner of understanding a test scenario, since the SUT inputs represent the test stimuli and the SUT outputs need to be evaluated so as to assess the behavior of the SUT.
The point is, however, that sometimes complex structures appear where the inputs and outputs cannot be separated. Hence, a few variants of a mixed version are allowed too. These are given in the statements (5.3 – 5.6).

The situation that the SUT interfaces of both directions need to be constrained in the *preconditions* part (cf. rules 5.3 – 5.4) happens when the tested SUT behavior can be activated only under certain circumstances that occurred before and led the SUT to a particular state. This state is then determined by the selected constraint on the SUT output.

$$IF\ constrained\_inputs_n \wedge constrained\_outputs_m \qquad THEN\ constrained\_inputs_n \wedge constrained\_outputs_m \tag{5.3}$$

$$IF\ constrained\_inputs_n \wedge constrained\_outputs_m \qquad THEN\ constrained\_outputs_m \tag{5.4}$$

$$IF\ constrained\_inputs_n \qquad THEN\ constrained\_inputs_n \wedge constrained\_outputs_m \tag{5.5}$$

$$IF\ constrained\_inputs_n \qquad THEN\ constrained\_outputs_m \tag{5.6}$$

The combination of interfaces of both directions in the *assertions* part (cf. rules 5.3 and 5.5) appears usually when some SUT output must be computed by application of the corresponding stimulus (e.g., calculation of the braking torque based on the position of the brake pedal).

Combining the different options in such a way that either only *constrained outputs* occur in the preconditions or only *constrained inputs* occur in the assertions is not allowed. The exclusive presence of the *constrained outputs* in the preconditions implies that some behavior had already happened before starting the considered scenario. By that, the scenario is dependent on some other scenario. That is not a convenient practice for the proposed TSpec algorithms since the test scenario should be possibly independent. Moreover, the test data cannot be generated automatically from such preconditions applying the currently available transformations. It also does not make sense to check the *constrained inputs* in the assertions since these are the SUT stimuli, not the SUT execution results.

The impermissible variants are listed in the formulas (5.7 – 5.9). They should be reformulated using *modus tollens*[35] or transposition[36] [Cop79, CC00] rules so as to adopt a valid form.

$$IF\ constrained\_inputs_n \wedge constrained\_outputs_m \qquad THEN\ constrained\_inputs_n \tag{5.7}$$

$$IF\ constrained\_outputs_n \qquad THEN\ constrained\_inputs_n \wedge constrained\_outputs_m \tag{5.8}$$

$$IF\ constrained\_outputs_n \qquad THEN\ constrained\_inputs_m \tag{5.9}$$

[35] In logic, modus tollendo tollens (Latin for "the way that denies by denying") [Cop79] is the formal name for indirect proof or proof by contraposition (contrapositive inference), often abbreviated to MT or modus tollens. It can also be referred to as denying the consequent, and is a valid form of argument (unlike similarly-named but invalid arguments such as *affirming the consequent or denying the antecedent*). Also known as an indirect proof or a proof by contrapositive [CC00]. Modus tollens has the following argument form:
If P, then Q.
¬Q
Therefore, ¬P.
Every use of modus tollens can be converted to a use of modus ponens and one use of transposition to the premise which is a material implication. For example:
If P, then Q. (premise - material implication)
If Q is false, then P is false. (derived by transposition).

[36] In the methods of deductive reasoning in classical logic, "transposition is the rule of inference that permits one to infer from the truth of "A implies B" the truth of "Not-B implies not-A", and conversely" [Cop79]. Its symbolic expression is: $(P \rightarrow Q) \leftrightarrow (\sim Q \rightarrow \sim P)$.

In the cases when only the interfaces of the same directions are constrained in both preconditions and assertions, respectively, as provided with the rules (5.10) and (5.11), the following holds. (5.10) can be restructured to the equivalent, more reasonable version given in (5.12) indicating that the test scenario should always be assessed, no matter what test stimuli have been currently applied, whereas (5.11) does not make sense at all since it does not test any behavior resulting from the SUT behavior.

$$\mathit{IF\ constrained\_outputs}_n \qquad \mathit{THEN\ constrained\_outputs}_m \tag{5.10}$$

$$\mathit{IF\ constrained\_inputs}_n \qquad \mathit{THEN\ constrained\_inputs}_m \tag{5.11}$$

$$\mathit{IF\ true\ {}^\wedge\ any\ constraints} \qquad \mathit{THEN\ constrained\_outputs}_n\ {}^\wedge\ \mathit{constrained\_outputs}_m \tag{5.12}$$

Additionally, the construction of IF-THEN rules can be enriched with the logical connectives and temporal expressions listed in Sections 4.1.3 and 4.1.4. Although not all of them have been implemented, a few of them deserve a special attention. These are:

- *OR* – for the considerations on the alternative [ETSI07]
- *during(x), after(y)* as expressions of temporal dependencies between the SigFs.

The former one is implemented as an extension of the arbitration mechanism, whereas the latter one is implemented as a workaround of the existing synchronization algorithms rather than dedicated structures.

The alternative given in (5.13) determines alternative behavior of the SUT. Whenever *A* holds, *B or C or D* should hold too. This statement is realized in MiLEST by an equivalent set of logical implications given in (5.14) providing that the arbitration mechanism is adjusted for this particular case.

$$\begin{array}{llll} \mathit{IF} & \mathit{A} & \mathit{THEN} & \mathit{B} \\ & & & \mathit{OR\ C} \\ & & & \mathit{OR\ D} \end{array} \tag{5.13}$$

$$\left.\begin{array}{llll} \mathit{IF} & \mathit{A} & \mathit{THEN} & \mathit{B} \\ \mathit{IF} & \mathit{A} & \mathit{THEN} & \mathit{C} \\ \mathit{IF} & \mathit{A} & \mathit{THEN} & \mathit{D} \end{array}\right\} \text{including the adjustment of the arbitration mechanism} \tag{5.14}$$

Here, all the cases are executed and checked in parallel and the snapshot known from the TTCN-3 alternative [ETSI07] does not have to be taken at all. Instead, it is enough to adjust the arbitration mechanism so as to add the semantic of the disjunction. Following the example from (5.13 and 5.14), if any assertion delivers a *pass* verdict, the entire logical implication passes as well, whatever verdict is provided by the remaining assertions.

Then, taking the expression *during(x)* as an example of temporal dependencies, the SF diagram enables to control the activation of the features in time by manipulating the signals. Considering the scenario provided in (5.15):

$$\mathit{IF} \quad \mathit{A} \quad \mathit{THEN} \quad \mathit{during(x)\ B,} \tag{5.15}$$

the extraction of SigF B takes place only *during* a certain period of time (here, *x seconds*). An example of the realization solution is given in Figure 5.15. The feature is checked only during time *x* multiplied by the current *time step*. Whenever the activation of SigF extraction appears, it happens in the restricted frames of time. To do so the *reset signal,* being a constituent of the activation signal, is constrained within the SF diagram (see Figure 5.16).

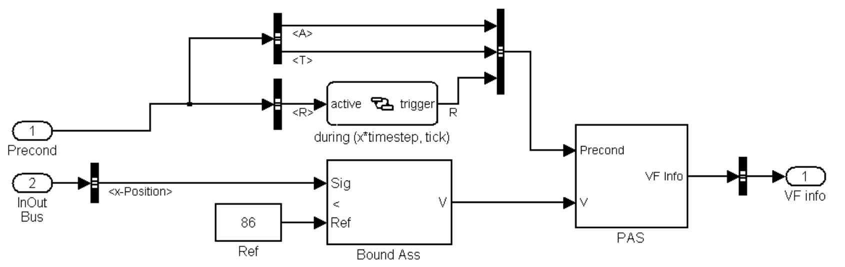


Figure 5.15: Implementation of during(x) TI feature – Feature Detection Level (Assertions).

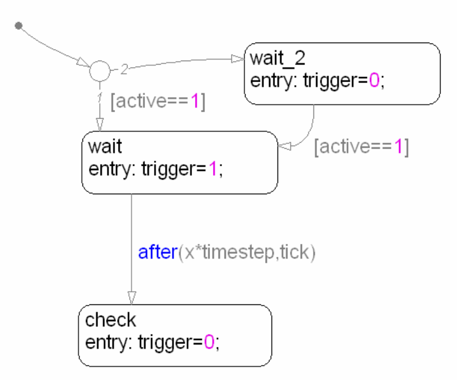
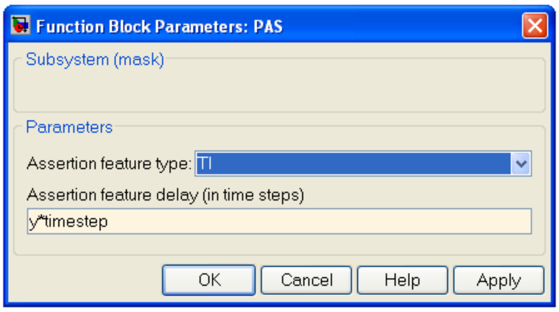


Figure 5.16: Insights of the SF Diagram for the Implementation of during(x) TI feature.

The expression *after(y)* refers to the logical implication given in (5.16). Here, the concept of features synchronizations may be re-applied so as to shift the activation of the features in time by manipulating the signals (cf. Figure 5.17). It is based on the retardation concept. The activation of SigF *B* extraction is retarded by application of a delay of *y seconds*.
If an identification delay for a given SigF already exists (see Section 4.2.1), it is summarized with the retardation caused by the temporal expression.

$$IF \quad A \quad THEN \quad after(y)\ B \tag{5.16}$$



Function Block Parameters: PAS

Subsystem (mask)

Parameters

Assertion feature type: TI

Assertion feature delay (in time steps)

y*timestep

Figure 5.17: Retarding the Extraction of TI Feature by Application of after(y).

An additional issue emerges when the SUT output signal is not a number (NaN) or is out of range. Although such a case is related to negative testing (cf. Section 8.1), which is beyond the scope of this work, the following solution is recommended and has been realized. An extra function checks all the output signals on violation w.r.t. the mentioned problems. If any of them is detected, the test execution is either aborted or paused with a clear indication on the faulty signal (see Section 5.5.2 for further explanation). Generally, it is good practice to shift such types of issues into the test control to achieve the separation of concerns splitting the functional abstract test scenarios from the run-time faults. Hence, it has been realized in the test control.

Whereas the process of test specification (TSpec) has already been described in many places in this thesis, the test data generation (TDGen) still deserves particular attention. This is due to (1) the automatic transformations applied to obtain the test stimuli, (2) generation of test signals variants, (3) their combination, and (4) sequencing of the obtained signals into the test steps, test cases, and test suites. Hence, the following three sections concern those issues in detail.

5.3 Automation of the Test Data Generation

The test development process proposed in this thesis can be automatized by application of transformations and the ready-to-use test patterns. This saves the test development time and enables the test engineers to focus on different aspects of the test coverage instead of the technical details.

The transformations allow for the retrieval of the test harness. The entire TDGen mechanism is supported. The test patterns help to collect the data, whereas the transformations serve for producing their variants systematically.

The application of automatic transformations assure that the TSpec and TDGen correspond to each other in a consistent manner. It is possible because the produced test signals are derived directly from the TSpec design.

5.3.1 Transformation Approach

Generation of the test data applies the currently known techniques in combination with the SigF concept. These are equivalence partitioning (EP) and boundary testing (BT). EP is a black-box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle, test cases are designed to cover each partition at least once.

Equivalence partitioning assumes that all values within any individual partition are equivalent for a selected test purpose [ISTQB06]. BT does not only select one value from the partition points, but boundary values. The test data variants are established for each SigF separately, based on a dedicated algorithm.
Basically, the problem of test stimuli generation can be divided into a set of problems classified according to different activities:

(1) Transformation:

- Generation of the structure for the test data based on the *preconditions* from *validation functions*
- Generation of the abstract SigFs generators

(2) Preparation of test signals generation:

- Establishing the ranges of the SUT input and output signals
- Establishing the partition points for every signal

(3) Generation of concrete test signals:

- Generation of signal variants based on the selected algorithms depending on the SigF type

(4) Combination of the obtained test signals and their sequencing:

- Combination of signal variants according to the chosen strategy for test automation
- Sequencing of the test signals over time in a test case or in a test suite
- Manual refinement if needed (e.g., when there exists a functional relation between the test cases)

The transformation is defined as a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another metamodel [CS03, BSK04]. The test data generation is realized using ML scripts, the metamodels are not directly necessary (cf. Section 5.8.2 of this chapter). The abstract objects of the SL/SF model are already available thanks to its comprehensive API.

5.3.2 Transformation Rules

The concrete high-level implementation rules applied in the prototype are listed below:

- Before the transformation begins, all necessary libraries must be loaded
- The link to the MiLEST library for all blocks added to the test model should be disabled in order to manipulate their parameters within this model
- For each SUT a test harness must be built
- The data types of all generated signals should inherit from the types specified for them in the SUT model and consequently in VFs
- The names of all generated Inport blocks at the source and signal lines connecting them with the SUT should match
- The structure of the *TestData* must be consistent with the *MIL_Test/Test Data/Test Data Architecture/<Test data generator>* from the library
- The simulation parameters in the resulting test model inherit the ones in the source model
- After a successful transformation the new test model is saved automatically

The technical transformation rules related to the specification levels are listed in Table 5.1 for illustration purpose. Starting at the test harness level, the TDGen unit is generated based on the structure of the TSpec unit. If the TSpec is further defined by the test engineer, the transformation functions refine the TDGen unit by updating the number of requirements and test data sets inside them. This is possible by analyzing the fixed elements of the TSpec, e.g., *‘Test Info’* interfaces, and projecting them onto the transformation target, e.g., *’Requirement’* subsystem in the TDGen. Similar methodology applies for all the levels in the test system hierarchy.

Table 5.1: Transformation Rules for Test Data Sets Retrieval.

Level	Test Specification	Test Data Generation
Test Harness	TSpec subsystem identified	Generate *’Test Data’* subsystem applying the pattern from *MIL_Test/Test Data/Test Data Architecture/<Test data generator>*
	Number of SUT input signals	Number of generated signals for *‘Test Data’* subsystem
Test Requirement	*’Test Info’* interface	Generate a *’Requirement’* subsystem
	Number of *’Test Info’* interfaces	Number of requirements
Test Case – Validation Function	*’Activate ’* interface	Generate a *’Test Data’* set pattern and a corresponding state in *’Generation Sequence’* diagram
	Number of *’Activate’* interfaces	Number of *’Test Data’* sets
	Number of *’Activate’* interfaces	Number of states in *’Generation Sequence’* diagram
Feature Generation – Feature Detection	SUT *input* signal in *Bus Selector*, ‘*MATLAB Fcn*’, signal comparison block	Generate a ‘*MATLAB Fcn’* connected to an output port labeled with SUT *input’s* name
	SUT *input* signal in *Bus Selector*, ‘*Logical expression*’, signal comparison block	Omit ‘*Logical expression*’ and detect other connected feature extractor
	SUT *input* signal in *Bus Selector*, ‘*Complete step*’, signal comparison block	Generate ‘*Complete step’* connected to an output port labeled with SUT *input’s* name
	SUT *input* signal in *Bus Selector*, ‘*Detect constant*’, signal comparison block	Generate a subsystem labeled ‘*Constant’* and an output port labeled SUT *input* signal
	SUT *input* signal in *Bus Selector*, ‘*Detect increase*’, signal comparison block	Generate a subsystem labeled ‘*Increase*’ and an output port labeled SUT *input* signal
	SUT *input* signal in *Bus Selector*, ‘*Detect decrease*’*,* signal comparison block	Generate a subsystem labeled ‘*Decrease*’ and an output port labeled SUT *input* signal
	SUT *input* signal in *Bus Selector*, ‘*Detect step’,* signal comparison block	Generate a ‘*Step*’ and an output port labeled SUT *input* signal
	SUT *output* signal in *Bus Selector*, signal comparison block and *reference* block	Generate a *Bus Selector* (with SUT *output* selected inside), signal comparison block, *reference* block, *Memory* block, and an output port labeled ‘*Activate*’
	SUT *input* signal in *Bus Selector* and signal comparison block parameterized by ‘==’	Generate a subsystem labeled ‘*Constant’* connected to an output port labeled with SUT *input’s* name
	SUT *input* signal in *Bus Selector* and signal comparison parameterized by ‘>’ or ‘>=’	Generate a subsystem labeled ‘*Increase’* connected to an output port labeled with SUT *input’s* name
	SUT *input* signal in *Bus Selector* and signal comparison parameterized by ‘<’ or ‘<=’	Generate a subsystem labeled ‘*Decrease’* connected to an output port labeled with SUT *input’s* name

The approach is summarized based on a simple, relatively abstract example in Figure 5.18. Assuming that a transformation from functional requirements into their conditional representation is already done, two VFs nested in the TSpec unit are designed (see the right part of Figure 5.18). The preconditions encapsulate information about the test data demanded to activate the appropriate assertions. For each single precondition a corresponding set of signal generators is used resulting in the test data sets. The next step is to constrain the data with time. Either default or parameterized duration time for a single signals set may be applied. A temporal constraint – *after(time1)* is used in the example below. Finally, further parameters (e.g., signal value, permitted value range) depending on the feature from which the corresponding signal is generated, are set. This is supported by the values contained in the precondition’s parameter *rs*.

Hence, the test signals sequences are obtained. In Figure 5.18, the preconditions correspond to their test data sets. If the transformation is complete, the generated signals activate every single assertion one after another following the predefined time intervals.

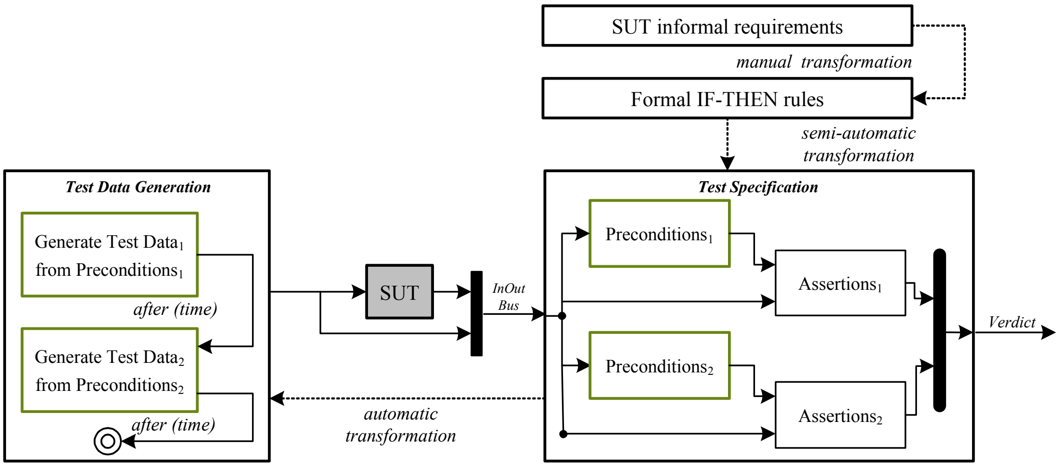


Figure 5.18: Test Stimuli Definition – an Abstract View.

In Figure 5.19, a similar situation as in Figure 5.18 is presented, but on the lower level of abstraction (i.e., using concrete signals). In *VF1,* all the values above the dotted line over the signal $u_1(t)$ activate the flag assertion. Thus, $u_1(t)$ is generated applying a corresponding pattern and it is available within time $\in (0, t_1)$. Afterwards, $u_1(t)$ remains unchanged and $u_2(t)$ *increases* within time $\in (t_1, t_2)$ so as to enable the flag assertion in *VF2.* In this example, no dependencies between features exist. Thus, the process of the test stimuli generation is completed.

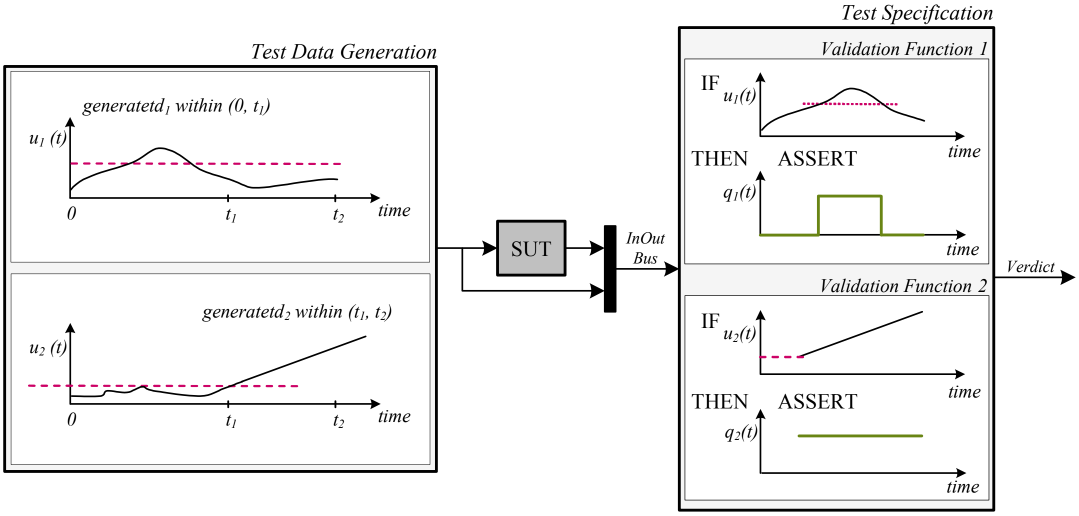


Figure 5.19: Test Stimuli Definition – a Concrete View.

5.4 Systematic Test Signals Generation and Variants Management

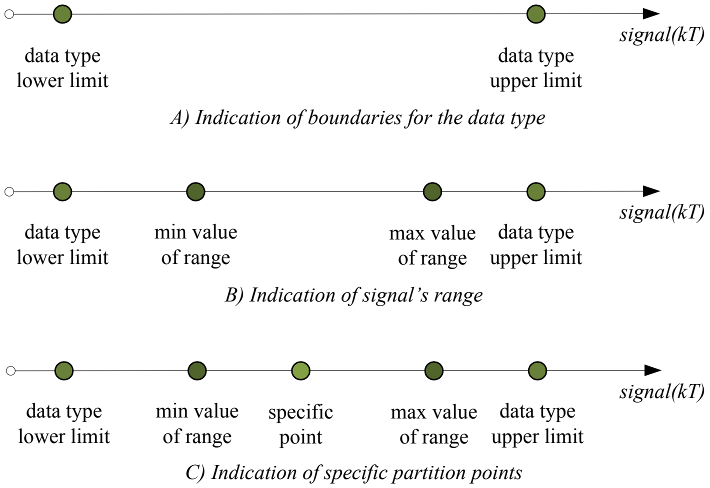
5.4.1 Generation of Signal Variants

The test data are the test signals which stimulate the SUT to invoke a given behavioral scenario. In the approach proposed here, a particular set of SigF generators produces selected signals for particular test cases.
The concrete variants of the test signals are provided based on the generation patterns discussed in Section 4.2. The variants generation method can be applied if the *signal ranges* and *partition points* have been defined on all the stimuli signals according to the requirements or engineer's experience.

Partitioning the range of inputs into groups of equivalent test data aims at avoiding redundant testing and improving the test efficiency and coverage. The situation is becoming even more complex when hybrid systems are considered since their signals vary continuously in value and time. However, in this work only values are partitioned, they relate to the signal characteristics. It is due to the fact that the SigF generation as such is already implicitly based on the time partitioning concept (cf. Figure 4.2). The duration time of a feature is taken as default unless no temporal expressions are included.

Practitioners often define *equivalent classes intuitively*, relying primarily on case studies. The success of this method depends on the tester's experience and his subjective judgments.
Another option is to *specify the equivalence* [Bur03] based on the requirements. This approach depends on the fact of whether the specification provides sufficient details from which the equivalence classes and boundaries could be derived.
At least three different methods can be used to choose the representatives of the equivalence class, namely *random testing*, *mean value testing*, and *boundary testing*.

Dedicated blocks, called signal range and partition points are provided for every SUT input and output interfaces in order to let the test engineer set the boundaries. Three types of such boundaries are distinguished. These result from the applied *data type,* the *signal range*, and *specific partition* points.
Data type boundaries are determined by the lower and upper limit of the data type itself (see Figure 5.20A). They are limited by its physical values. For example, the lower limit of temperature is absolute zero (i.e., 0 K or -273.15˚ C); an unsigned 8-bit value has the range from 0 to 255.
The *range* of the signal belongs to the data type range and it is specific for the SUT (see Figure 5.20B). For example, if water is the test object, water temperature is the input to the SUT; test data for water may be specified to be between 0˚ C and 100˚ C.
Finally, the *partition points* (see Figure 5.20C) are of concern since they constitute the specific values of critical nature belonging to the signal range.



A) Indication of boundaries for the data type

B) Indication of signal's range

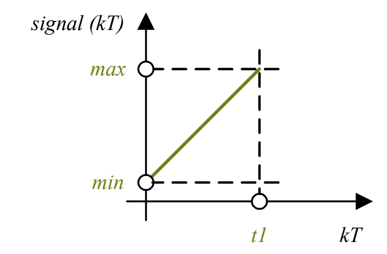
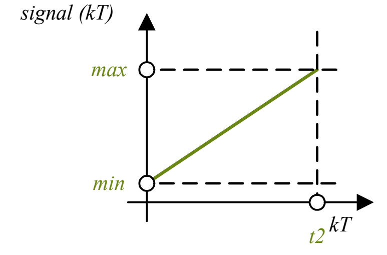
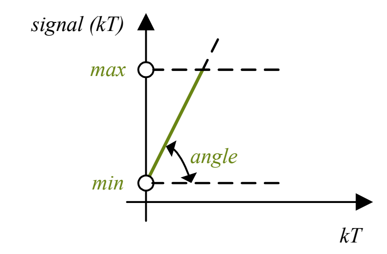
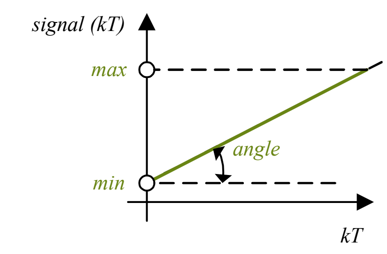
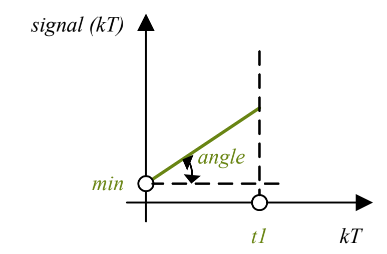
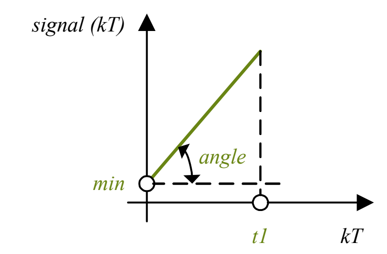
C) Indication of specific partition points

Figure 5.20: Steps of Computing the Representatives.

A number of algorithms are proposed for signal variants generation depending on the SigF type. The analysis of SigF types, equivalence partitioning and boundaries are used in different combinations to produce the concrete test data variants.

As an example, the feature *increase* is considered in the following. A number of generation options given in Table 5.2 are possible to produce the signal variants systematically. Here, three possibilities showing two variants for each are considered. For reasons of simplicity the ramp is selected as a shape representative of an *increase*. In the first option, *timing constraint* and *signal range* are the factors indicating the generation rule. Hence, two different ramps are obtained, both covering the entire range and preserving the proper time constraint (*t_1* and *t_2* respectively). In the second option, *signal range* and the *tangent of the angle* play a significant role. Thus, it does not matter how long the signal is generated, the signal variants must hold within the given tan(angle) along the entire value range. Finally, in the third option, *timing constraint* and *tangent of the angle* determine the generation rule. At this point only one default boundary of the *signal range* is considered, the *tan(angle)* is preserved and the predefined *t_1* indicates the duration of a signal generation.

Table 5.2: Options for Increase Generation.

no.	*Options:*		*variant 1*	*variant 2*
1	*Criteria:*			
	time	+		
	signal range	+		
	tangent (angle)			
2	*Criteria:*			
	time			
	signal range	+		
	tangent (angle)	+		
3	*Criteria:*			
	time	+		
	signal range			
	tangent (angle)	+		

Considering the *increase* generation in terms of a real-world signal, vehicle velocity is taken as an example. Its value range is between <-10, 70>. Additionally, {0} is taken into account since the car changes its driving direction from backwards to forwards at this point. The third generation option is chosen from Table 5.2. The algorithm computes 10% of the current range around all boundaries and partition points. Thereby, variant$_n$ of a signal$_m$ belongs to the range calculated according to the formula given in (5.17) for lower or upper limits, respectively, where p is a partition point or a boundary point.

$$<p_n, p_n+10\% \cdot (p_{n+1}-p_n)> \text{ or } <p_n-10\% \cdot (p_n-p_{n-1}), p_n> \tag{5.17}$$

Hence, the following *increases* are obtained as representatives: <-10,-9>, <-1,0>, <0,7> and <63,70>. The duration of those *increases* can be either derived from the VFs, or set as default values, or changed manually. The steps of computing the representatives on the value axis are illustrated in Figure 5.21.

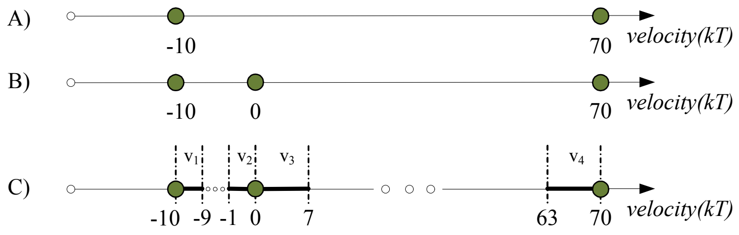


Figure 5.21: Steps of Computing the Representatives for Vehicle Velocity.

Firstly, the boundaries of the range are identified (step A) resulting in a set {-10, 70}. Then, all risk-based partition points are included (step B) resulting in value {0}. They are derived from the specification or the test engineer's experience. The *increase* ranges are calculated and the variants in those ranges are generated (step C) as given below. Finally, durations of the features are added on the time axis.

$v_1 \in <-10, -10+10\% \cdot (0-(-10))> \equiv <-10, -9>$
$v_2 \in <0-10\% \cdot (0-(-10)),0> \equiv <-1, 0>$
$v_3 \in <0, 0+10\% \cdot (70-0)> \equiv <0, 7>$
$v_4 \in <70-10\% \cdot (70-0),0> \equiv <63, 70>$

The example discussed above is an instance of an *explicit partition*, when a single signal is involved in the partitioning process. Additionally, *implicit partitions* appear when the SUT input signals are put in a mathematical relation with each other. In such a case, both sides of the equation (or inequality) are considered. Taking the inequality given below as an example, generation of variants for the relation of two SigFs A and B, on the left, depends on the variants for SigF C, and vice versa.

$A – B > C$

Hence, firstly the right-hand side partitions for C are produced. Then, the variants for A and B on the left-hand side are generated in such a way that the inequality for the representatives of C is valid. Later on, the procedure is repeated for the left-hand side SigFs, whereat the redundant cases are deleted.

In MiLEST not only *SUT input partitioning*, but also *SUT output partitioning* [FDI$^+$04] is applied. This enables (1) to verify the power of the equivalence partitions built for the SUT input signals and (2) to improve the test data generator. The procedure is similar to the input partitioning. This time, the ranges of SUT output signals are defined and their partition points are identified. Alternatively, the type of SigFs constraining the output in the assertions may be taken into account. Then, after the test execution, it is checked whether the expected results of the test cases cover all possible equivalence partitions of the SUT output. If this is not the case, additional test stimuli are designed so as to cover the missing values.

5.4.2 Test Nomenclature

Summarizing the approach in terms of the nomenclature given in Sections 5.1 – 5.4, the following definitions are provided. The generated test signals create the behavior of a *test case*. *Test* *case* is a set of input values, execution preconditions, expected results, and execution postconditions, developed for a particular test objective[37] so as to validate and verify compliance with a specific requirement [ISTQB06].

A *test step* is derived from one set of VFs preconditions. It is related to the single scenario defined in the VF within the TSpec unit. Thereby, it is a basic, non-separable part of a test case. The *test case* can be defined as a sequence of *test steps* dedicated for testing one single requirement. Obviously, if only one VF is defined for a given requirement within a TSpec, the *test step* corresponds to the *test case*. Otherwise, a *test case* consists of as many *test steps* as VFs exist for a single requirement. Hence, the number of test cases is the same as the number of requirements multiplied by the maximal number of variants constructed for the feature generators within this single requirement.

W.r.t. the architecture of the test system, the *test step* corresponds to the so-called *Test Data set*, whereas the *test case* is composed of a *sequence of such test steps* within one single requirement block. This means that the *test case* is constructed by the *Generation Sequence* block and *Test Data* blocks. By that, the traces to the requirements are implicitly obtained.

A *test suite* is a set of several test cases for a component or SUT, where the postcondition of one test is often used as the precondition for the next one [ISTQB06]. The specification of such dependencies takes place in the test control unit in the proposed test framework.

The concept of a test suite according to the definition given in [ISTQB06] is particularly important in the context of integration level test, where the test cases depend on each other. Otherwise, if no relations are noted, a test suite is simply a collection of ordered test cases.

5.4.3 Combination Strategies

When a test involves multiple signals, which is usually the case, the combination of different signal variants should be established. In the proposed framework, the combination is done at the test case level. In particular, the generated variants of SigFs in one test step (i.e., in one *Test Data* set block) are combined. Technically, this is possible by manipulating the numbers passing to the switch present at the feature generation level so as to control the application of feature generators. Several combination strategies to construct the test cases are known, e.g., *minimal combination, one factor at a time*, and *n-wise combination*[38] [LBE$^+$04, GOA05]. Combination strategies are used to select a subset of all iterations of different variants of test signals based on some coverage criterion. In the following, three combination strategies are discussed. However, the implementation attached to this thesis realizes the first two only.

Minimal combination, denoted as A++B means that each class in A and B is considered at least once [LW00, CDP$^+$96]. It iterates every interface and ends with the last variant. This one is held until other input variants iteration ends. Figure 5.22 illustrates the iterations based on three inputs. Only two iterations are obtained because every input has two variants. Each of them appears once in a combination. The error detection coverage is not satisfactory for this method, though.

[37] Test objective is a reason or purpose for designing and executing a test.

[38] Further combinations strategies are *random combination*, which does not support reliable test coverage [CDP$^+$96, LBE$^+$04, Con04a] or *complete (maximum) combination*, which leads to exhaustive testing.

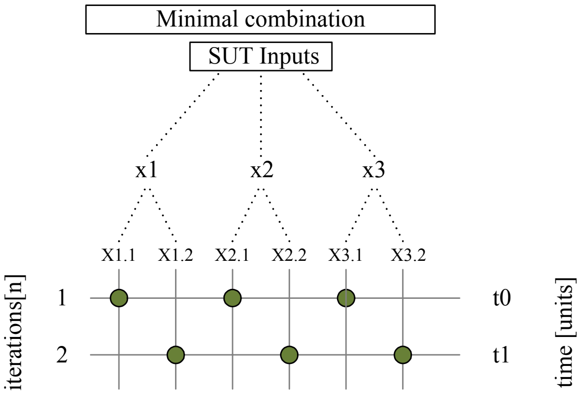


Figure 5.22: Minimal Combination.

One factor at a time method uses a default normal condition as the starting point. Then, in the next iteration only one parameter at a time is changed, under the assumption that there is no interaction between parameters. Figure 5.23 presents the iterations based on this method.

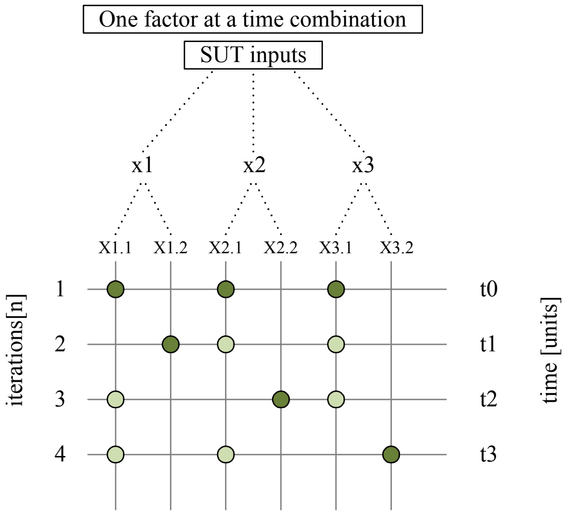


Figure 5.23: One Factor at a Time Combination.

For *n-wise combination* [CDP+96, LBE+04, GOA05] every possible combination of n classes is selected at least once. A special case of n-wise combination is a *pair-wise combination*. Here, o*rthogonal arrays* [Man85, GOA05] may be applied. Figure 5.24 illustrates a situation when the iterations are constructed by 2-wise combination. Three inputs with two variants match the orthogonal arrays. Compared to the 'one factor at a time method', this one computes the same amount of iterations. Nevertheless, the iterations based on the orthogonal array a stronger ability to find errors [CDP+96].

n	Column		
	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

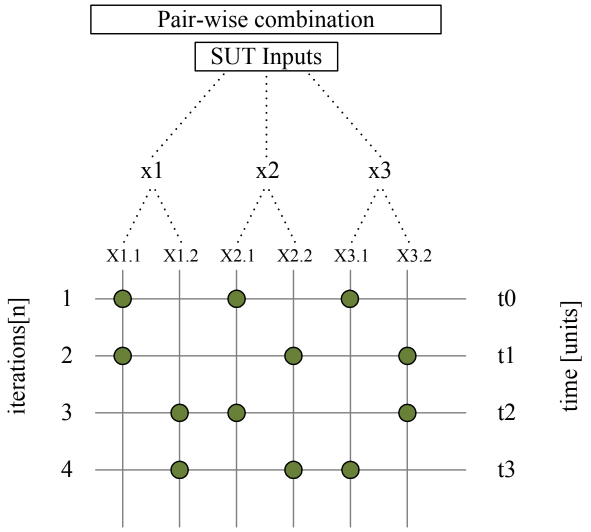


Figure 5.24: Pair-wise Combination.

5.4.4 Variants Sequencing

If the test data variants are calculated and the combination strategy has been applied, the test cases can be established. Every set of SigF generators constitutes a test purpose for a test case. All the sets together form a test suite and are sequenced in the test control. Before the details of the test control are discussed, the sequencing of the test signals within the test data sets will be explained. Every single generated signal is activated for a given predefined period of time. The SigFs within one set need to be synchronized so that the timing is the same for all of them. The SF diagram on the test data level called *'Sequencing of test data in time due to Preconditions' (Std)* controls the activation of a given variants combination applying a predefined duration time.

If there is no temporal constraint within the *preconditions* in a VF, the following timing issues apply for the test data sequencing:

- *Duration time of a test case* execution (TCD) is equal to the duration time of the test data set execution multiplied by the number of test data sets assuming that the duration time of the test data set is fixed, otherwise the duration time of the test data sets are summarized for a given test case.
- *Duration time of a test data set* generation includes the duration time of the selected variant generation for a given set of SigFs and the transition time to the next test data set (if any exists).
- *Execution time of the entire test suite* for a *single combination* of variants is equal to the sum(TCD).
- *Execution time of the entire test suite* including *all the combinations* of variants (i.e., for the entire test design) is equal to the sum(TCD) multiplied by the maximum number of variants in all test data sets.

- Maximum number of all test data variants in one test data set for the analyzed test suite indicates the *number of loops over the test control*.
- The *maximum number of variants* can be calculated from the analysis of the number of variants for every single test data set. The maximum found for any precondition is denoted as the maximum number of variants.
- *Sequencing of the test data variants* can be controlled either by the SF diagram or by the iteration number applied currently in the test control loop. In the former case all the variants are applied one after another until all of them have been executed. In the latter case, they are applied in a functional sequence determined by the test suite (i.e., test cases sequence given in the test control). If the number of loops is higher than the number of variants in a particular test data set, then the last available variant is used over and over again.

In the case when temporal expressions appear in the preconditions of VFs, they should be included in the calculation, extending the duration time of SigFs generation, respectively.

The activation of test data sets must be synchronized with the timing given in the test control algorithm. Thus, taking *minimal combination* of variants as an example, in *Std* a time-related parameter is added. It results from the way the test control is specified. It enables the starting point of a selected test data set to be synchronized forming a test case on the test data level with the starting point of a test case within the test control.
The first test case in the test control starts without any delay, so the parameter should be equal to 0. The duration of this first test case specified on the test control level determines the starting point of the next test case. Hence, the starting of the following test case appears after the former finishes, which means after a specified period of time. The same applies to the activation of the test data sets on the test data level. It starts after the same period of time as the test case specified on the test control level. Further on, the next following test case starts after all the previous ones finish. Thus, the same applies to the activation of the further test data set. Since the duration of all the previous test cases on the test data level is not explicitly included within the *Std*, it is calculated by summarizing the durations of all previous test cases.

An example illustrating this algorithm is given in Figure 5.25. Assuming that a test control presented on the left-hand side is given, the parameters are calculated as shown on the arrows provided in the middle of Figure 5.25.

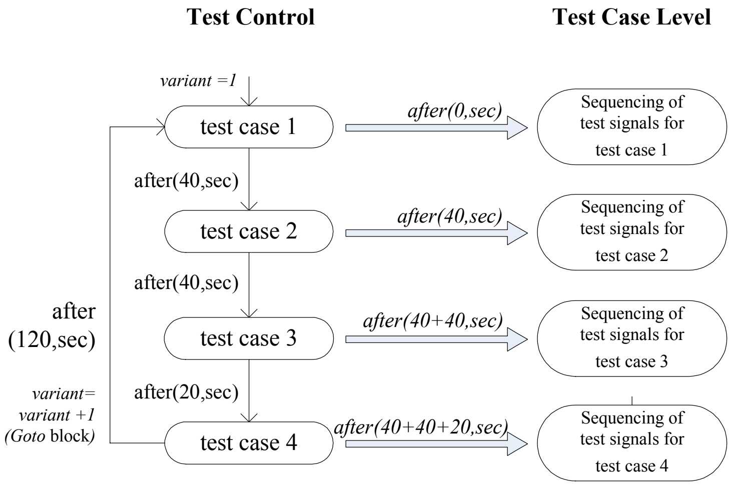

Test Control
Test Case Level
variant =1
test case 1
after(0,sec)
Sequencing of test signals for test case 1
after(40,sec)
test case 2
after(40,sec)
Sequencing of test signals for test case 2
after(40,sec)
test case 3
after(40+40,sec)
Sequencing of test signals for test case 3
after(20,sec)
test case 4
after(40+40+20,sec)
Sequencing of test signals for test case 4
after (120,sec)
variant= variant +1 (Goto block)

Figure 5.25: Test Control and its Implication on the Test Data Sequencing.

This algorithm applies only to the first iteration within the test control. If more iterations are needed (i.e., if more variant sets are applied for a particular test suite execution), a more complex algorithm should be used depending on the number of iterations.

The test cases are executed one after another according to the sequence specified in the test control. After execution of one set of variants for such a sequence the next set of variants is chosen and consequently, the sequence including new test stimuli repeats.

5.5 Test Reactiveness and Test Control Specification

Test control is a specification for the invocation of test cases within a test context. Test configuration is determined by the chosen SUT, the components, the initial parameters that must be set to let this SUT run and a concrete test harness.

The test control in a traditional meaning is a concept widely known in the protocol testing, but also discussed for testing automotive system [Con04b]. In this thesis, it is considered as a means to achieve reactive testing. [Leh03] defines the test reactiveness as a reaction of the TDGen algorithm on the SUT outputs during the test execution. In particular, the stimulation mechanism in a test case reacts on a defined SUT state, instead of on a defined time point.

In this thesis, the definition of reactive testing is extended [Zan07]. Additionally, the TDGen algorithm can be controlled by signals from the test evaluation system. Hence, not only can the test cases' execution be organized over time, but also the data generation may be influenced by the verdict of the previous test case (as in [ETSI07]); the SUT outputs (as in [Leh03]) and by other test evaluation signals (e.g., reset, trigger, activation). Loops and conditions can be used to specify the execution order of individual test cases [ETSI07]. Depending on how the test control is defined, it enables the inclusion/exclusion of only those test cases, which are/are not of our interest at a particular moment.

Moreover, the signal generation becomes more flexible and supports major SUT changes without needing to be parameterized again after each SUT update.

Similarly to [Syn05] and [Leh03], a reactive test is defined as a test that is able to react to the SUT behavior and adapt to the newly observed situation dynamically within one simulation step during the execution. The test system is required to run synchronously to and simultaneously with the SUT model so that test actions can be performed using the same concept of time.

If during a test run an identifiable behavior is observed, then the test control decides on the further test execution, i.e., if a strategic test case fails, the current test control algorithm is activated. A test may either be aborted, provide a warning, invoke/redefine another desirable test case, or change the sequence and range of the applied test data, etc.

5.5.1 Test Reactiveness Impact on Test Data Adjustment

The process of TDGen can be impacted by different factors. The design principles listed in Table 5.3 refer to different reactiveness scenarios that enhance an automatic TDGen. A name of a method indicates its application context, whereas an example is a solution proposal.
Principle no. 1.1 has been identified in [Leh03]. The TDGen depends on the SUT outputs as discussed by [Leh03]. A set of non-reactive test cases for different SUT variations can be specified as a single reactive test suite that automatically adapts itself to the actual variation. Principle no. 1.2 enhances systematic concrete test data retrieval by parameter adjustment. Principle no. 1.3 refers to the situation when the internal signals of the test evaluation influence the test data.

Table 5.3: Test Data Generation Dependencies.

no.	Test Data Generation – Design Principles	Influenced by
1.1	*Name:* Use an SUT output value to compute the test data. *Example:* If an SUT output reaches certain value, then perform further action within the TDGen algorithm. This concept is realized by [Leh03].	SUT Output
1.2	*Name:* Use verdict value for parameter sweep within the TDGen process. *Example:* If verdict of test case X = none, then change the value of a particular parameter within the test data (e.g., the signal range) to meet the appropriate coverage of signal range. Note that the last value of the refined parameter must be stored to make the test repeatable.	Verdict
1.3	*Name:* Compute the temporal dependencies of SigF generation within the TDGen process. *Example:* If trigger signal *T* appearing in the test evaluation unit indicates that a single feature has already been assessed, then start generating a new variant of this feature (e.g., complete step generation) at this particular time point.	Termination of the validation process for a selected SigF

A classic example of the application of principle no. 1.3 application is the generation of different step functions for the measurement of the step response characteristics. If the signal generation algorithm knows when the SUT output has stabilized and a verdict has been established, it neither outputs a second step too early – eliminating the possibility of an incorrect measurement

– nor too late – preventing wasting important testing time. Moreover, the signal generation becomes flexible and supports major SUT changes without needing to be parameterized again after each SUT update [MP07]. For example, every single step function represents one *variant* of a step.
The test engineer decides about the arrangement of test signals or test cases using predefined constraints. For that purpose a number of predefined generic conditions that may be applied by a tester to constrain the test control, is provided. These are:

- if *signal* =, <, <=, >=, >, ~= *value*
- if *signal* =, <, <=, >=, >, ~= *value @ time* (e.g., if *signal = value @end time of a test*)
- if *signal = value @ [*$time_1$*, *$time_2$*]* || *(*$time_1$*, *$time_2$*)* || *[*$time_1$*, *$time_2$*)* || *(*$time_1$*, *$time_2$*]*
  where: @ – at,
  || – means logical or.

The *signal* can be replaced by a concrete instantiation:

- *local verdict* or *overall verdict* – value range ∈ {pass; fail; none; error}
- *evaluation trigger*, *reset signal* – value range ∈ {1,0}
- *SUT output signal* – value range ∈ {flows, features, strings, numbers, etc.}.

The resulting examples are, respectively:

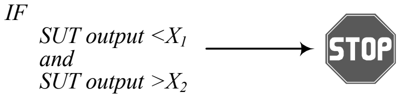
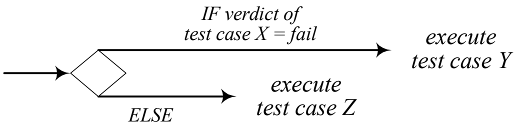
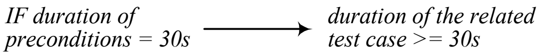
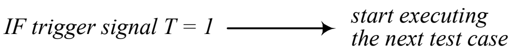
- if *evaluation trigger = 1*
- if *local verdict = none* at *6th second*
- if *local verdict = pass* in the interval between *(3,5) seconds.*

5.5.2 Test Control and its Relation to the Test Reactiveness

In Table 5.4 a number of factors influencing the test control and different scenarios including those factors are introduced. The scenarios are denoted as test control principles. They refer to different reactiveness paths and support an automatic execution of a test. They are elicited based mainly on the experience gained by testing the adaptive cruise control (ACC). Principle 2.1 presents the case when the SUT output range overflow is used to decide about the further progress of the test execution. If an SUT output involved in a test case appears to be out of an allowed range, the execution of this test case should be stopped or paused (cf. Section 5.2). Additionally, at least all the test cases where this particular signal is involved in the validation process should not be executed before the SUT is fixed.

As previously mentioned, a name of a principle indicates its application context, whereas an example is a solution proposal.

Table 5.4: Test Control Principles on the Component Test Level.

no.	Test Control – Design Principles	influenced by
2.1	*Name:* Check if an SUT output is outside a value range and reacts to the conclusion in a proper way. *Example:* If an SUT output is out of an allowed range, then stop executing this test case and do not execute all the test cases where this particular signal is involved in the validation. 	SUT output
2.2	*Name:* Use verdict value to control the order of test cases. *Example:* If verdict of a test case X = fail, then execute test case Y, else execute test case Z.  IF verdict of test case X = fail execute test case Y execute test case Z ELSE Note that this principle relates to the traditional understanding of the test control.	Verdict
2.3	*Name:* Use temporal constraints from preconditions to indicate how long a given test data set should be generated. *Example:* If preconditions X demand 30 seconds, then test case X' should last at least 30 seconds. This should be specified in the test control.  IF duration of preconditions = 30s duration of the related test case >= 30s Note that this principle relates to the way of specifying the test control rather than to the reaction of the test system.	Temporal constraints in the validation process for a selected SigF
2.4	*Name:* Use validation signals to compute the test case starting time. *Example:* If trigger signal *T* appearing in the test evaluation indicates that a feature has already been assessed within a considered test case, then start a new test case at this time point.  IF trigger signal T = 1 start executing the next test case	Termination of the validation process for a selected SigF

As previously mentioned the SUT outputs (principle 2.1), verdicts (principle 2.2), and evaluation signals (principle 2.4) might impact the test execution. Furthermore, temporal constraints specified in the functional requirements may indicate the duration time of particular test cases or duration time of a test step as explained in Table 5.4, point 2.3. The duration time of a particular test case can be calculated according to formula (5.18):

$$\textit{test case duration} = \sum_{i=1}^{i} (x + \textit{SigF specific time})_i \tag{5.18}$$

where:

- *SigF-specific time is the stabilization time of a SigF,*
- *x equals time specified in the temporal expression,*
- *i is the number of test data variants applied for a given test case.*

Note that the SUT output signals influencing the test arrangement may be interpreted either directly by the test control unit, or indirectly by the validation functions in terms of verdicts. This issue is left open since the test designer decides where, when, and which option to use. There exist cases favoring one of these approaches depending on the complexity of the SUT and the resulting TSpec.

5.5.3 Test Control Patterns

The traditional test control patterns can be realized by the SF diagram. An example of variant-dependent test control is shown in Figure 5.26.

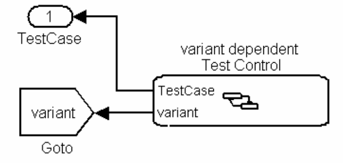


Figure 5.26: Variant-Dependent Test Control.

The insights of the test control are illustrated in Figure 5.27. In this example, a test suite including four test cases is provided. Every test case, apart from the very first one, is activated after a given period of time, which is equal to the duration time of the previous test case. Furthermore, the applied variants combination can be controlled. Here, *test case 1* is activated first and when the entire test suite has been executed, the next one follows.

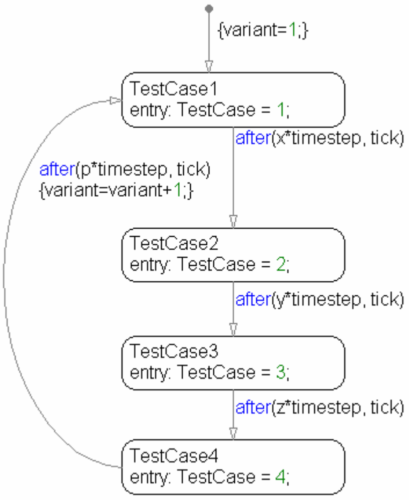


Figure 5.27: Test Control Insights for Four Test Cases.

Additionally, in Figure 5.26, the number of currently applied variant is forwarded into *Goto* block. Such a design of the test control enables to set the corresponding variants on the test data level. There, a block called *From* (cf. Figure 5.9) receives the variant number from the *Goto* block.

The test control patterns facilitated by the reactive testing concept are usually application specific. Thus, the realization example and an explanation will follow in Section 6.3.

5.6 Model Integration Level Test

Analyzing the integration level test the cross-cutting nature of functionality and failures occurrence is considered. This enables quality assurance (QA) to be approached in terms of interacting services, which is a potential candidate for dealing with the integration problems [EKM$^+$07]. Service-orientation means, in this context, a design paradigm that specifies the creation of automation logic in the form of services. A *service* itself is a cross-cutting functionality that must be provided by the system. Like other design paradigms, it provides a means of achieving a separation of concerns [All06]. The *services* are sometimes also referred to as system functions or functionalities in the automotive domain. Services identify partial behaviors of the system in terms of interaction patterns [Krü05, EKM06, BKM07]. Based on the interactions between services the test specifications are developed.

5.6.1 Test Specification Design Applying the Interaction Models

The analysis of the system starts at the requirements level, where features and functions of new software within a car [KHJ07] are described. The requirements are used as a basis for interaction models. These models present the behavior between different software parts. In the following the extended Message Sequence Charts (MSCs) [ITU99, Krü00] combined with the hybrid Sequence Charts (*hySC*) [GKS99], called *hySC for testing (hySCt)*, are applied as a modeling technique. With this practice, both discrete and continuous signals exchanged between the components, can be expressed. Such *hySC* can be advantageously used in the early phases of the development process, especially, in the requirements capture phase. The interacting components are specified as the roles and the functional relations between them are the services.

In the upcoming paragraphs the syntax and some semantics extensions of [Krü00] (as compared to traditional MSCs [ITU96, ITU99] combined with the semantics given by [GKS99] are applied. In particular, (1) *arrows* to denote events are used. Then, (2) *angular box* denotes conditions on the component's variables suggested by [GKS99]. A pair consisting of an arrow and an angular box defines (3) the *conditions* (i.e., states defined by a set of signal feature (SigF)) being mainly a result of sending the information continuously. This enables the notation for continuous behavior to be clarified. However, the semantics remains the same as [GKS99] defined: (4) a continuous *global clock* exists and an abstract time axis is available for each component. The components occurring in the sequence charts are connected by channels along which information exchange occurs. It is possible that more than one signal (i.e., message) appears simultaneously (5). The dashed vertical lines are used to denote *simultaneous signals* appearance. A simple example is given in Figure 5.28. Also, (6) *triggers*, (7) parallel *signal receiving*, (8) an *alternative*, and (9) *local timers* are available. These are specified in [ITU99]. It is left open which concrete modeling dialect should be used in the future.

The resulting *hySCts* are translated to the MiLEST syntax and semantics. Hence, the *hySCts* are transformed to the VFs and interpreted in terms of their execution.

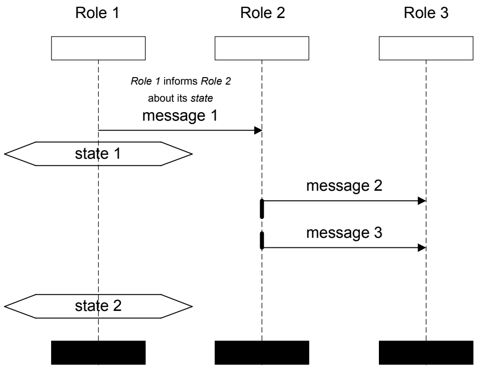


Figure 5.28: Basic hySCt.

The *hySCts* are the interaction models representing the services as shown in Figure 5.28. A service is defined by the interaction of its roles. In an interaction, a *role* represents the participation of a system entity in an interaction pattern. Between roles, different information can be exchanged. These are: a message, an event, a continuous signal, or a reference to another service. Compound interaction elements consist of a number of interaction elements composed by operators. Instances of operators are sequence, loop, alternative, parallel, and join, representing sequential composition, repetition, choice, parallel, and join composition, respectively [EKM06]. The state in an angular box indicates a condition and remains unchanged until a new state appears. Please note that this interaction model abstracts from concrete notations – in the context of this work a combination of selected modeling dialects has been used.

HySCts combined with the SigF paradigm introduced in Chapter 4 of this thesis enable the flow of information (i.e., signals, events, messages) between different components (i.e., roles) to be designed without specifying how this flow should be realized and how the components process the inputs to produce valid outputs. This abstract design method is powerful enough for specifying the integration level test. Further on, the resulting elements are incorporated into and executed together with the SUT model.

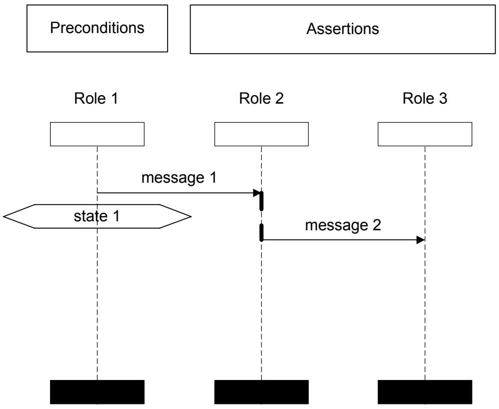


Figure 5.29: HySCt including VFs Concepts.

In Figure 5.29, a simple example of a *hySCt* enriched with additional elements is given. In reference to the test specification, *preconditions* and *assertions* boxes being the constituents of a VF are added at the top of the diagram so as to identify which information should be transformed to the appropriate VF part. The dashed lines indicate the simultaneity.

Then, it is aimed to facilitate the retrieval of the running VFs from the *hySCts*. Hence, the boxes called preconditions and assertions present at the *hySCts* diagrams are reused. They recall the notion of the test specification. The role of each such VF, in this context, is to detect the SUT failures. Every *hySCt* may result in one or more VFs depending on the complexity of the requirement on which it was built and its own complexity.

5.6.2 Test Data Retrieval

At this point such a test specification can be utilized twofold: (1) as an information source for the test data generator for further test design refinement; (2) to evaluate the SUT behavior during the test execution (being the primary aim of a VF existence).

Since (2) has already been discussed in the previous sections of this thesis (cf. Section 4.4), herewith more attention will be given to (1).

Applying the algorithms for test stimuli derivation as described in Sections 5.3 – 5.5 the redundant test cases would be obtained. The redundancy (i.e., in the sense of duplication) is caused by generating similar test data for different VFs because of the existence of similar precondition sets. Additionally, it may happen that several VFs include exactly the same sets of pre-preconditions, resulting from the *hySCts*, next to their original preconditions. This situation occurs especially at the system integration level test when the previous test scenarios play a significant role for the current test scenario. In that case, the test objective being checked at the moment may only be reached when the SUT is brought to a predefined state.

Hence, as a solution to this problem, not only the VF's preconditions, but also higher-level *hySCts (HhySCts)* decide on the contents of generated test data sets. *HhySCts* are defined similarly as high-level MSCs in [ITU99, Krü00]. They indicate sequences of, alternatives between, and repetitions of services in two-dimensional graphs – the nodes of the graph are references to *hySCts*, to be substituted by their respective interaction specifications. *HhySCts* can be translated into basic *hySCts* without loss of information. Generally, the rule applies that only those VFs, where the entire sequence of interactions within one single service is caught, should be considered for the test data generation. If this is the case the number of the test data sets does not match the number of VFs. In fact, the number of test data sets is less than the number of VFs and it matches the number of services present in the higher-level *hySCt*. This is an advantageous situation in terms of testing as all possible behavioral sequences are still covered and the tests are not redundant.

Additionally, the sequences of the generated sets of the test data are obtained from the higher-level *hySCts* by analyzing the paths of those charts are analyzed. The algorithm for obtaining possibly lots of the meaningful sequences results from the white-box test criterion commonly known as *path testing* [ISTQB06].

Further on, when the test data sets are set and their sequencing algorithms have been defined, the automatic test data variants generation and their combination methods may be applied without any changes as described in Sections 5.4 – 5.5.

5.6.3 Test Sequence versus Test Control

In this thesis, at the component level test, the single test case consists of several test steps. The test control allows for specification of the execution order for such test cases including their different variants. Hence, no additional concept for a test sequence is used.

At the integration level test the situation changes. Here the *test sequence* is derived from the *HhySCts*. It includes a set of test cases that need to be executed in the order specified by this sequence. Then, the *test control* enables to manage all the resulting test sequences and their variants. An example of such circumstances will be given in Section 6.4.

5.7 Test Execution and Test Report

The test execution does not demand any further effort other than the simulation of any SL model. The test assessment is already included in the test design, thus the verdicts are immediately obtained. This is possible due to the existence of the test oracle and arbitration mechanism in the TSpec unit. Additionally, the quality metrics for evaluating the test model are calculated and the test report is generated after the test execution (see Chapter 7).

Although producing the reports is rather an implementation issue, its basic design has been provided in this work and reports have been generated for the case studies presented in Chapter 6. The MATLAB® Report Generator™ [MRG] and Simulink® Report Generator™ [SRG] are used for defining and generating compact, customizable documents automatically after test execution. They include the applied test data, their variants, test cases, test control, test results, and the calculated quality measures. The presentation form is constituted by a number of tables, plots of signals, graphs, verdict trajectories, and their textual descriptions [Xio08]. The generic design of a report template enables it to be adapted to the actual test model and SUT configuration. Technically, both report generation and test system need to be coordinated. Many functions contribute to the report contents depending on the available models.

Before generating the test reports, the test results have to be recorded in the ML workspace. Also, the verdicts need to be ordered according to the delays they were identified with during the test so as to provide the test results in a correct manner.

5.8 Related Work

The model-based test approaches have already been reviewed in Chapter 3. Similarly, related work on properties of signals has been discussed in Section 4.5. However, a few issues deserve a special attention since they either contribute to the results achieved in this thesis or constitute the ongoing efforts that are based on the experience gained here.

5.8.1 Test Specification

Time Partitioning Testing (TPT) [Leh03] is the primary example using the concept of SigF in an evaluation system. Although the test assessment is mainly offline – due particularly to the real-time constraints at the HiL level – TPT concepts are a basis for the TSpec unit described in this thesis.

Similarly, the Classification Tree Method [GG93] illustrates how to construct test stimuli systematically. Even if the realization proposed in MTest [Con04b] enables test data to be only created manually, the concept constitutes the fundamental principles according to which the test data generator embedded in TDGen unit is built.

Also, the synchronization mechanism for the test evaluation system of MiLEST realized in [MP07] is a considerable contribution to this thesis. As discussed in Chapter 3, the transformations of [Dai06] contribute to the MiLEST automation. [SCB, EmbV] help to conceive of the TSpec. [CH98, WCF02] underline the test evaluation problems. [SLVV] gives the technological solution for building the prototype of MiLEST.

Further on, the progress achieved in MiLEST is the foundation for the ongoing efforts towards UML Testing Profile for Embedded Systems (UTPes) [DM_D07, Liu08].

5.8.2 Transformation Possibilities

The background knowledge for the MiLEST transformations has been gained from the experience on Model Driven Architecture (MDA) artifacts [MOF] applied in the context of testing [ZDS$^+$05, CBD$^+$06, GCF06, Dai06]. This practice enabled the creation of generic transformation functions that may be applied to any model[39].

The reasoning on the MDA-related transformation for testing proposed by [DGN04, Dai04, Dai06] may be followed for the framework provided in this thesis in an analogue way. Figure 5.30 shows a layered metamodel hierarchy applicable to MiLEST. It has been resigned from developing the detailed metamodels for both ML/SL/SF and MiLEST since the direct ML-based solution appears to be feasible and performs well. The continuation of this topic, e.g., along the application of Query/View/Transformation (QVT) techniques is, however, not excluded from the future research as indicated in [ZSF06].

[39] This is possible under the assumption that the guidelines given in Section 5.3 are followed.

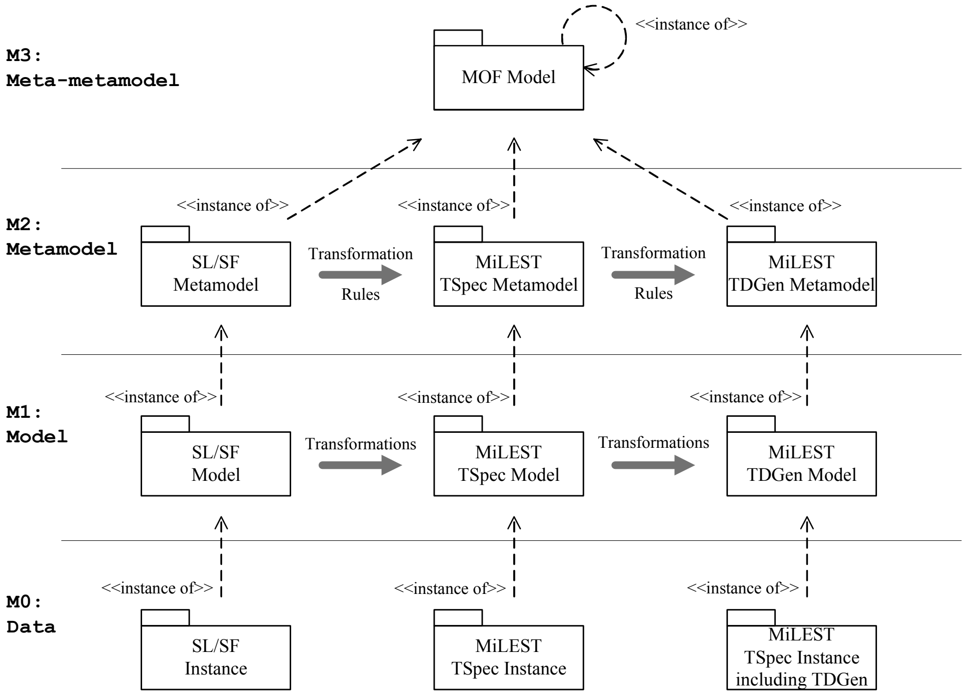


Figure 5.30: The Metamodel Hierarchy of MiLEST.

5.9 Summary

In this chapter, the concepts of testing in MiLEST have been introduced and explained following its development phases. Technically, MiLEST extends and augments SL/SF for designing and executing tests. It bridges the gap between system and test engineers by providing a means to use the SL/SF language for both SUT and test specifications, similarly to the way [Dai06] did for the Unified Modeling Language [UML] world. Moreover, it allows for the reuse of system design documents for testing. By that, the test development is early integrated into the software production.

This chapter adresses the third set of the research questions given in the introduction to this thesis as well. It has been illustrated that the test development process can be automated by application of test patterns and transformations. In particular, the test data and their systematically selected variants can be generated automatically from the formerly designed test specification. Then, the test control patterns can be applied. The manual workload during the test design phase cannot be fully excluded, though. The concrete VFs must be added by the test engineer, even if many hierarchically organized patterns for the test specification ease this process considerably.
Regarding the test execution, the test evaluation runs automatically on the fly, which allows for an immediate analysis of the test results. Test reports are obtained automatically as well.

In particular, in this chapter, the classification of signal features presented in Chapter 4 has been recalled to describe the architecture of the test system. In Section 5.1, different abstraction levels of the test system have been provided. They were denominated by the main activities performed at each level. The test harness level includes patterns for TSpec, TDGen, and test control. Then, the test requirements level has been followed by test case and validation function levels. Afterwards, the feature generation and feature detection have been elaborated on. In Appendix C, an overview of all the hierarchy levels is given. Also, different options for the specification of a test have been reviewed, revealing the challenges and limitations of the test design. The importance of the test evaluation has been emphasized. Furthermore, principles for the automatic generation of test data have been presented. By means of the concrete generic transformation rules, the derivation of test signals from the VFs has been formalized. Similarly, the generation of signal variants has been investigated. Combination strategies for test case construction have been outlined and sequencing of the generated variants at different levels has been reported. The concepts of reactive testing and of test control have been summarized too.

Section 5.6 presents considerations on the integration level testing. An advantage of using the ML/SL/SF framework is the possibility to execute both the system and test models. Thus, in Section 5.7, the test execution and test reporting have been discussed.

Finally, in Section 5.8, related work on test design, transformation approaches, and ongoing work towards UTPes [DM_D07, Liu08] have been elaborated.

– Part III –
MiLEST Application

6 Case Studies

"To every action there is always opposed an equal reaction."

- Isaac Newton

In this chapter, an analysis of the Adaptive Cruise Control (ACC) [Con04b] is presented. Its requirements and some further details are provided in Section 6.1. The ACC controls the speed of the vehicle while maintaining a safe distance from the preceding vehicle. ACC divided into different functional units forms more case studies. By that, three examples are elicited: pedal interpretation, speed controller, and ACC. They demonstrate the application of the concepts presented in the previous chapters of this thesis according to the Model-in-the-Loop for Embedded System Test (MiLEST) method.

The test development process is illustrated for all the case studies; however, every time another test aspect is investigated. For the first two examples, a similar presentation scheme is followed, although the emphasis is put on different steps of the process. Testing the pedal interpretation component gives insights into the specification of concrete validation functions (VFs), test data generation algorithms for them and the test control arranging the obtained test cases applying the minimal combination strategy. In the test specification for the speed controller, the VFs are defined as first. Then, the test reactiveness concept on the level of test data and test control is exploited. In the third example, the details of testing at the model integration level are additionally provided. Here, besides the VFs, specific interaction models are provided and the test sequences in relation to the test control are explicitly considered.

Regarding the structure of this chapter, in Section 6.2, the pedal interpretation as an instance of an open-loop system is introduced. There, the main concepts of the test data generation algorithms are discussed. In Section 6.3, the speed controller as a representative of a closed-loop electronic control unit (ECU) is investigated. In this part, the attention is given to the test control and test reactiveness. Then, in Section 6.4, the model integration level test concepts are reviewed in the context of ACC functionality. Section 6.5 finishes this chapter with a summary.

6.1 Adaptive Cruise Control

The ACC controls the speed of the vehicle while maintaining a safe distance from the preceding vehicle. There are two controllers within the ACC: a speed controller and a distance controller. Both operate in a loop in conjunction with the vehicle.

The speed controller of an ACC measures the actual vehicle speed, compares it with the desired one and corrects any deviations by accelerating or decelerating the vehicle within a predefined time interval. Afterwards, the vehicle velocity should be maintained constant, if the desired speed does not vary.

If the vehicle approaches a car traveling more slowly in the same lane, the distance controller recognizes the diminishing distance and reduces the speed through intervention in the motor management and by braking until the predefined 'safe' distance has been reached. If the lane is clear again, ACC will accelerate to the previously selected desired speed.

When the deceleration performed by ACC is not sufficient because another car suddenly cuts out in front, ACC requests the driver through acoustic signals to additionally apply the brakes manually as [Con08] specifies. If the speed drops below 11 m/s because of the traffic, ACC will automatically turn off. In Table 6.1 a set of ACC requirements is given in a more condensed manner.

Table 6.1: Selected Requirements for Adaptive Cruise Control.

ID	*Requirements on ACC*
1	The ACC controls the speed of the vehicle while maintaining a safe distance from the preceding vehicle. There are two controllers within the ACC: a speed controller and a distance controller.
2	The speed controller measures the actual vehicle speed, compares it with the desired one, and corrects any deviations by accelerating or decelerating the vehicle within a predefined time interval.
3	If the desired velocity is considerably changed, the controller should react and adapt the vehicle velocity. This happens within a certain time due to the inertial characteristics of the velocity, which is related to the vehicle dynamics.
4	Afterwards, the vehicle velocity should be maintained constant, if the desired speed does not vary.
5	If the vehicle approaches a car traveling more slowly in the same lane, the distance controller recognizes the diminishing distance and reduces the speed through intervention in the motor management and by braking until the predefined 'safe' distance has been reached.
6	If the lane is clear again, ACC accelerates to the previously selected desired speed.
7	If the deceleration performed by ACC is not sufficient because another car suddenly cuts out in front, ACC requests the driver through acoustic signals to additionally apply the brakes manually.
8	If the speed drops below 11 m/s because of traffic ACC, automatically turns off.
9	If the braking action is undertaken, the ACC should switch off and the car should brake (velocity should decrease) as long as the braking pedal is pressed.
10	If the ACC has been active before braking, it should not be reactivated when the braking action is stopped.
11	If acceleration action is undertaken by a driver, the car should speed up.
12	If the ACC has been active before the acceleration, it should be reactivated when the acceleration action is stopped.
13	The ACC can be activated when the velocity is higher than 11 m/s.

A realization of the ACC provided by Daimler AG is demonstrated in Figure 6.1. Different components are responsible for different functionalities. Here, the loop between the ACC and a vehicle can be observed. Also, the pedal interpretation component used later in Section 6.2 is present there.

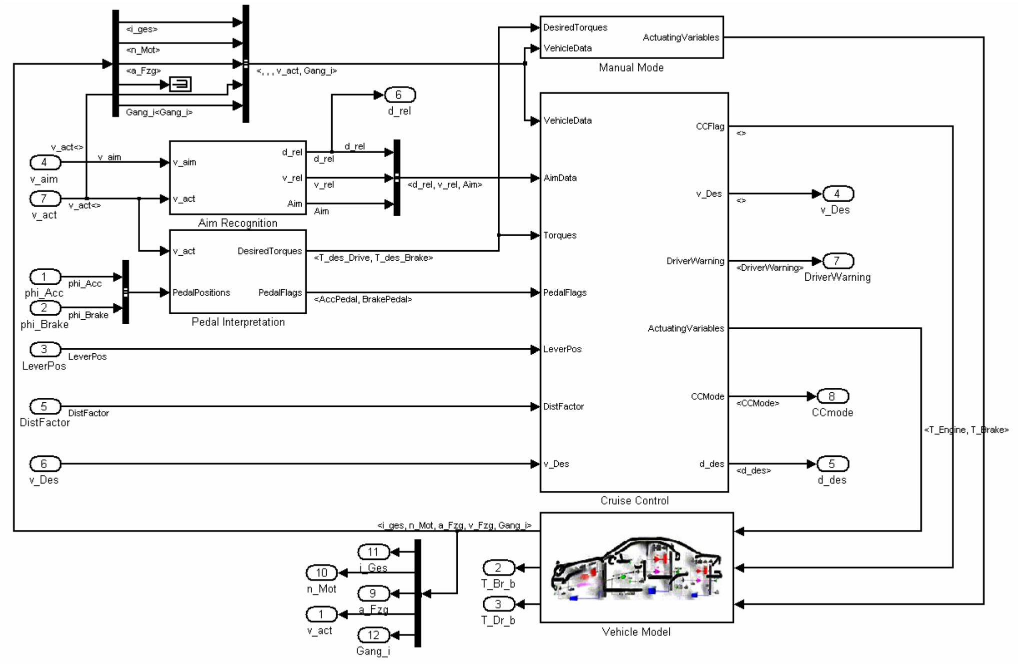


Figure 6.1: Components of the ACC System.

In Figure 6.2, the insights of the *real* cruise control are shown. The two controllers and some helping coordinators are specified. The speed controller being tested in Section 6.3 is illustrated at the left top of the figure.

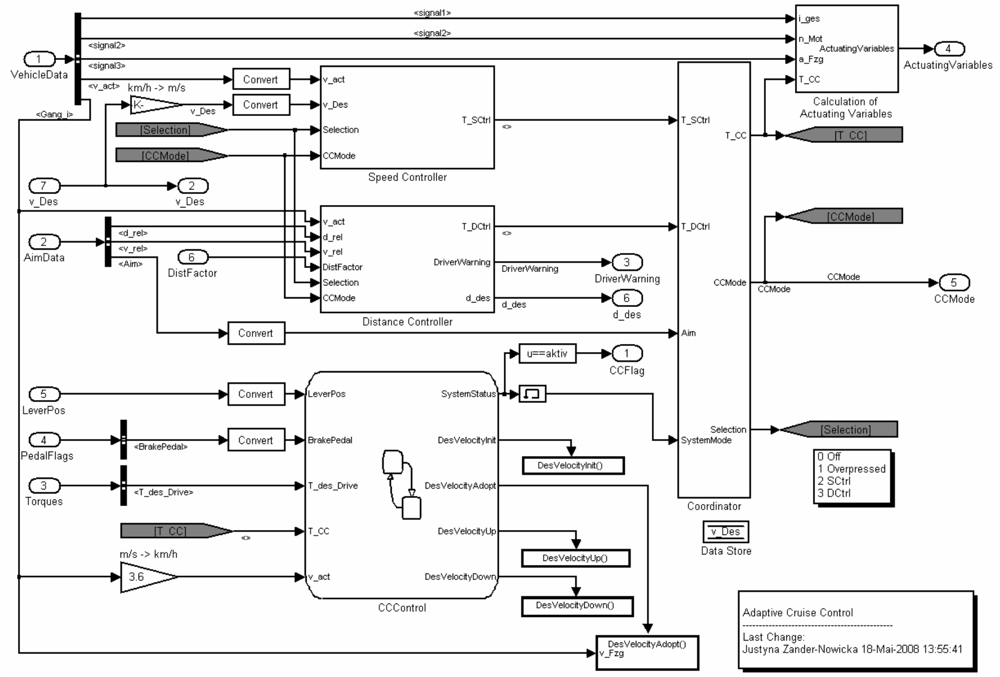


Figure 6.2: Components of the Cruise Control Subsystem.

6.2 Component Level Test for Pedal Interpretation

This and the two following sections demonstrate the application of MiLEST concepts. Testing the pedal interpretation component illustrates the process of VFs specification based on the selected system requirements. Also, test data generation patterns and their corresponding variants generation algorithms are given. Finally, the test control arranging the resulting test cases by means of the minimal combination strategy is introduced.

A simplified component of the pedal interpretation of an ACC is being tested. This subsystem can be employed as pre-processing component for various vehicle control systems. It interprets the current, normalized positions of acceleration and brake pedal (phi_Acc, phi_Brake) by using the actual vehicle speed (v_act) as desired torques for driving and brake (T_des_Drive, T_des_Brake). Furthermore, two flags (AccPedal, BrakePedal) are calculated, which indicate whether the pedals are pressed or not. Some excerpts of its functional requirements are given in Table 6.2, while the SUT interfaces are presented in Table 6.3 and in Table 6.4.

Table 6.2: Requirements for Pedal Interpretation (excerpt).

ID	*Requirements on pedal interpretation*
1	**Recognition of pedal activation** If the accelerator or brake pedal is depressed more than a certain threshold value, this is indicated with a pedal-specific binary signal.
1.1	**Recognition of brake pedal activation** If the brake pedal is depressed more than a threshold value ped_min, the BrakePedal flag should be set to the value 1, otherwise to 0.
1.2	**Recognition of accelerator pedal activation** If the accelerator pedal is depressed more than a threshold value ped_min, the AccPedal flag should be set to the value 1, otherwise to 0.
2	**Interpretation of pedal positions** Normalized pedal positions for the accelerator and brake pedal should be interpreted as desired torques. This should take both comfort and consumption aspects into account.
2.1	**Interpretation of brake pedal position** Normalized brake pedal position should be interpreted as desired brake torque T_des_Brake [Nm]. The desired brake torque is determined when the actual pedal position is set to maximal brake torque T_max_Brake.
2.2	**Interpretation of accelerator pedal position** Normalized accelerator pedal position should be interpreted as desired driving torque T_des_Drive [Nm]. The desired driving torque is scaled in the non-negative range in such a way that the higher the velocity is given, the lower driving torque is obtained[40].

Table 6.3: SUT Inputs of Pedal Interpretation Component.

SUT Input	**Velocity (v_act)**	**Acceleration pedal (phi_Acc)**	**Brake pedal (phi_Brake)**
Value Range	<-10, 70>	<0, 100>	<0, 100>
Unit	m/s	%	%

[40] A direct interpretation of pedal position as motor torque would cause the undesired jump of engine torque while changing the gear while maintaining the same pedal position.

Table 6.4: SUT Outputs of Pedal Interpretation Component.

SUT Output	Acceleration pedal flag (AccPedal)	Brake pedal flag (BrakePedal)	Driving torque (T_des_Drive)	Braking torque (T_des_Brake)
Value range	{0, 1}	{0, 1}	<-8000, 2300>	<0, 4000>
Unit	-	-	Nm	Nm

6.2.1 Test Configuration and Test Harness

The test configuration for the pedal interpretation SUT is straightforward, since it is an open-loop system. The insights into the pedal interpretation are provided in Figure 6.3.

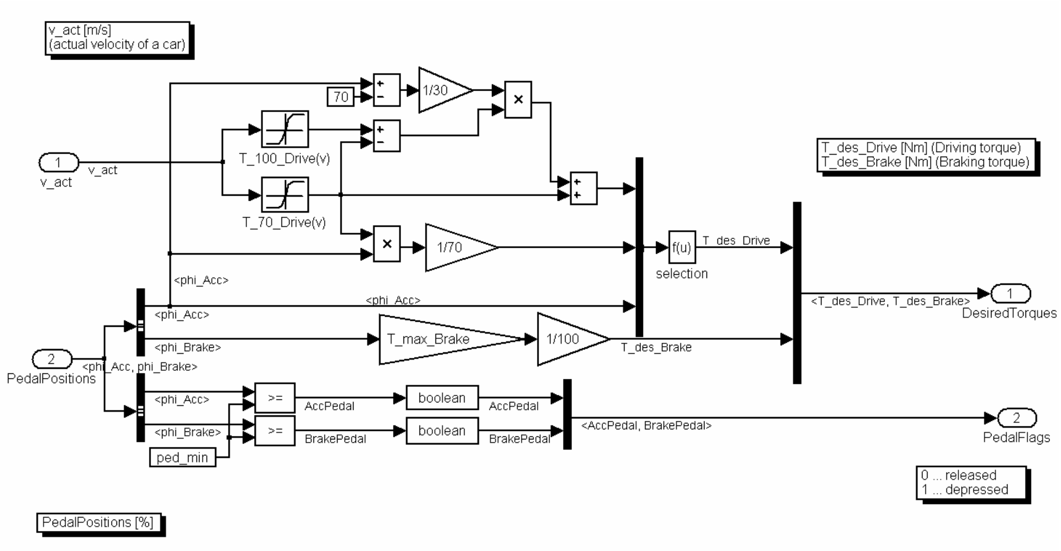


Figure 6.3: Pedal Interpretation Insights.

When the SUT is elicited from the entire ACC functionality, the test harness is built automatically around it (see Figure 6.4). Then, further refinements of the test specification are needed.

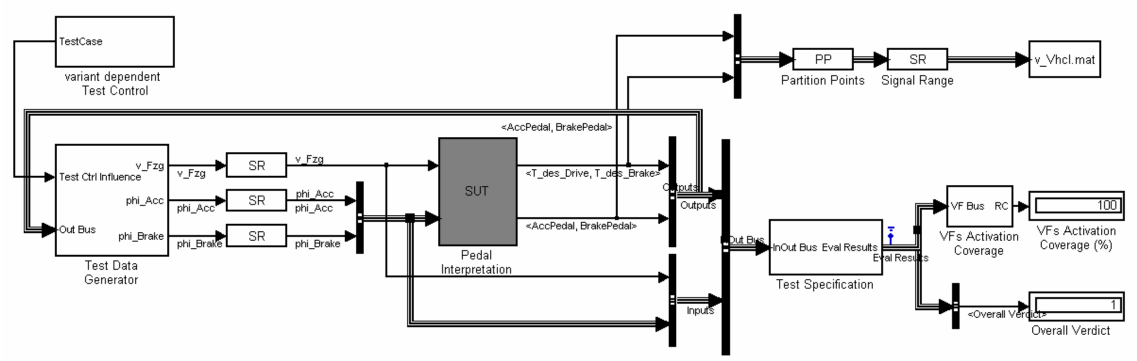


Figure 6.4: The Test Harness around the Pedal Interpretation.

6.2.2 Test Specification Design

The design of the test specification includes all the requirements of the pedal interpretation listed in Table 6.2. By that, four meaningful test sub-requirements (i.e., 1.1 – 1.2, 2.1 – 2.2) emerge. These result in the validation functions (VFs). Requirement 2.2 is analyzed for illustration purposes. The following conditional rules are based on the VFs provided here:

- IF *v* is constant AND *phi_Acc* increases *AND T_des_Drive* is non-negative
  THEN *T_des_Drive* increases.
- IF *v* increases AND *phi_Acc* is constant *AND T_des_Drive* is non-negative
  THEN *T_des_Drive* does not increase.
- IF *v* is constant *AND phi_Acc* decreases *AND T_des_Drive* is non-negative
  THEN *T_des_Drive* decreases.
- IF *v* is constant *AND phi_Acc* decreases *AND T_des_Drive* is negative
  THEN *T_des_Drive* increases.
- IF *v* is constant AND *phi_Acc* increases *AND T_des_Drive* is negative
  THEN *T_des_Drive decreases*.
- IF *v* is constant AND *phi_Acc* is constant
  THEN *T_des_Drive* is constant.

The VFs for the formalized IF-THEN rules are designed as shown in Figure 6.5. The actual signal-feature (SigF) checks are done in assertions when they are activated by preconditions.

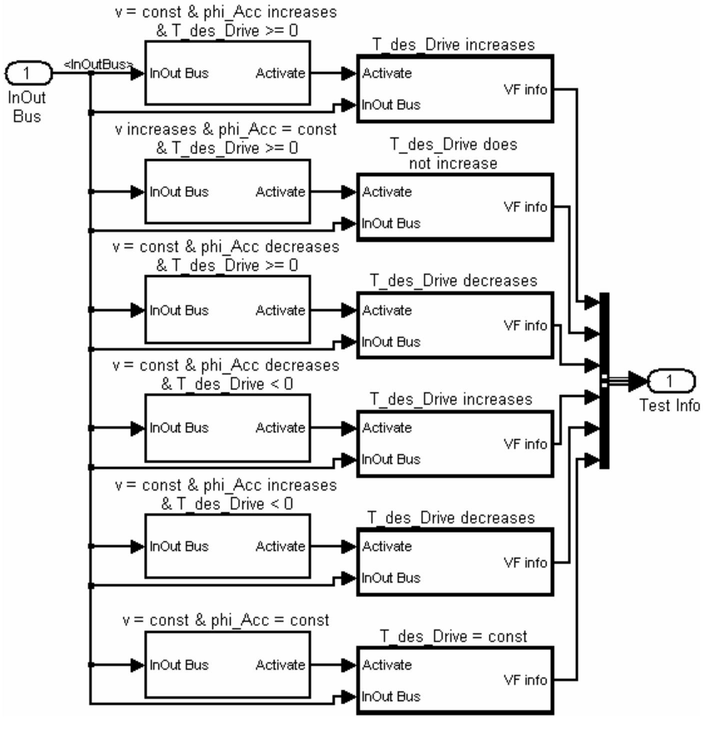

1
InOut Bus
<InOutBus>
v = const & phi_Acc increases & T_des_Drive >= 0
T_des_Drive increases
v increases & phi_Acc = const & T_des_Drive >= 0
T_des_Drive does not increase
v = const & phi_Acc decreases & T_des_Drive >= 0
T_des_Drive decreases
v = const & phi_Acc decreases & T_des_Drive < 0
T_des_Drive increases
v = const & phi_Acc increases & T_des_Drive < 0
T_des_Drive decreases
v = const & phi_Acc = const
T_des_Drive = const
InOut Bus
Activate
VF info
1
Test Info

Figure 6.5: Test Specification for Requirement 2.2.

An insight into a VF is given for the first one from Figure 6.5: If the velocity is constant and an increase in the acceleration pedal position is detected as illustrated in Figure 6.6, then the driving torque should increase as given in Figure 6.7.

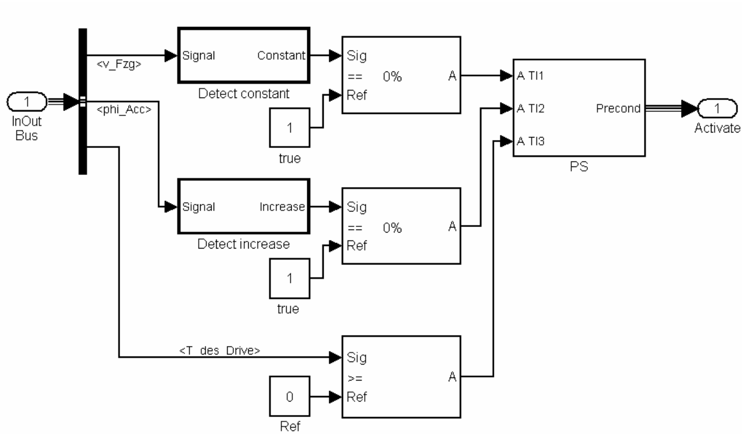


Figure 6.6: Preconditions Set: v = const & phi_Acc increases & T_des_Drive >= 0.

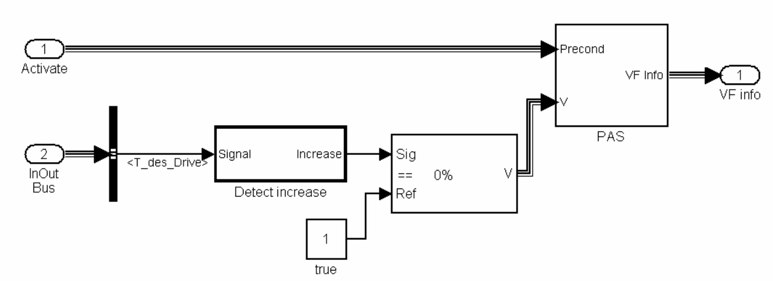


Figure 6.7: Assertion: T_des_Drive increases.

6.2.3 Test Data and Test Cases

When all the VFs are ready and the corresponding parameters have been set, test data can be retrieved. Using the preconditions from Figure 6.5 and the patterns for test data generation discussed in Section 5.1, the design given in Figure 6.8 is automatically obtained as a result of the transformations. Then, the test data generator (TDG) is applied to derive the representative variants test stimuli.

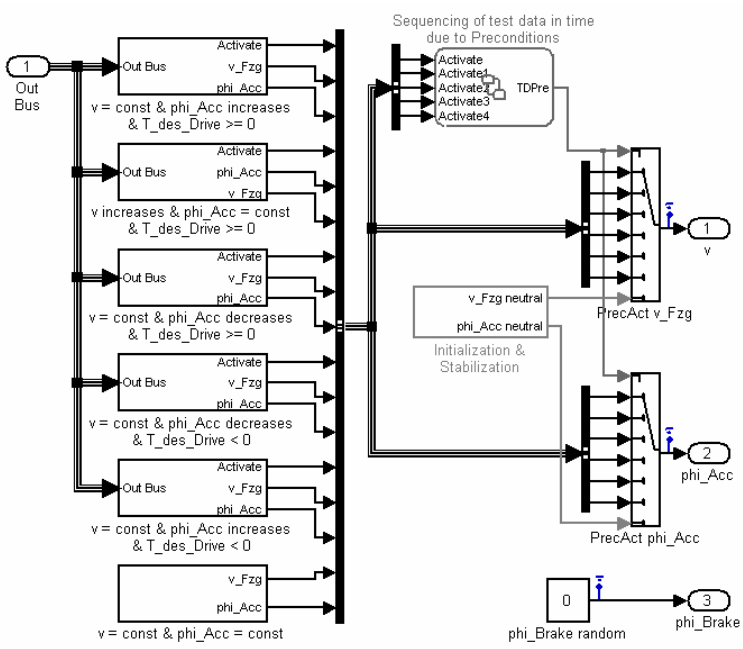


Figure 6.8: Derived Data Generators for Testing Requirement 2.2.

The number of preconditions blocks in Figure 6.5 suits the number of VFs appearing in Figure 6.8. Sequencing of the SigF generation is performed in the Stateflow (SF) diagram. Signal switches are used for connecting different features with each other according to their dependencies as well as for completing the rest of the unconstrained SUT inputs with user-defined, deterministic data, when necessary (e.g., *phi_Brake*).

Thus, as shown in Figure 6.9 (middle part) a constant signal for velocity is generated; its value is constrained by the velocity limits <-10, 70>. The partition point is 0. The TDG produces five variants from this specification. These belong to the set: {-10, 5, 0, 35, 70}.

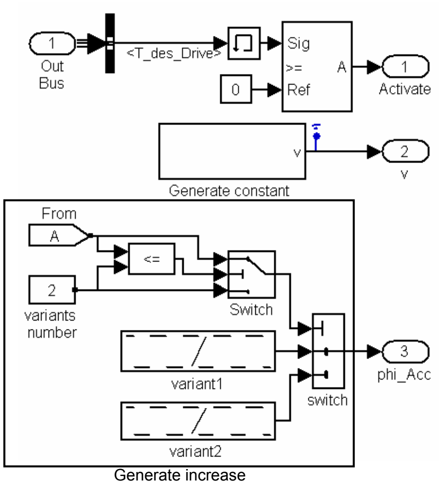
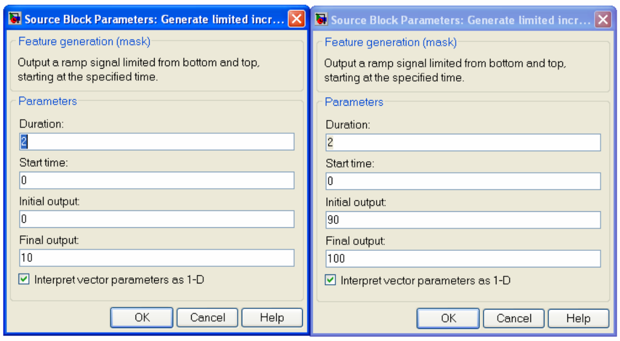


Figure 6.9: Test Data for one Selected Precondition Set.

For the acceleration pedal position limited by the range <0, 100> an *increase* feature is utilized. Furthermore, it is checked whether the driving torque is non-negative. This is the condition allowing the generation of the proper stimuli in the final test execution. The entire situation is depicted in Figure 6.9 (bottom part).

The *Generate increase* subsystem is shown to illustrate the variants generation. Here, two variants of the test data are produced. These are the *increases* in the ranges <0,10> and <90,100>. They last 2 seconds each (here, default timing is used). The brake pedal position is arbitrarily set since it is not constrained by the preconditions. Then, the combination strategy is applied according to the rule: If the current number of the variant is less than the maximal variant number, the switch block chooses the current number and lets it be the test signal variant, otherwise the variant that is last in the queue (i.e., maximum) is selected.



Source Block Parameters: Generate limited incr…

Feature generation (mask)

Output a ramp signal limited from bottom and top, starting at the specified time.

Parameters

Duration: 2

Start time: 0

Initial output: 0

Final output: 10

☑ Interpret vector parameters as 1-D

Source Block Parameters: Generate limited incr…

Feature generation (mask)

Output a ramp signal limited from bottom and top, starting at the specified time.

Parameters

Duration: 2

Start time: 0

Initial output: 90

Final output: 100

☑ Interpret vector parameters as 1-D

Figure 6.10: Parameterized GUIs of Increase Generation.

6.2.4 Test Control

The insights into the test control are shown in Figure 6.11. Since there are no functional relations between the test cases, they are ordered one after another using the synchronous sequencing algorithm for both SigF generation and test cases. The default duration of SigF at the feature generation level is synchronized with the duration of a corresponding test case at the test control level. Technically, this is achieved by application of *after(time$_1$, tick)* expressions.

Moreover, there is a connection of variants activation on the test data level with the test control level. It happens along the application of the *From* block deriving the variant number from the *Goto* block specified on the test control level as discussed in Section 5.5. Here, the context of minimal combination strategy of variants is applied at both test data and test control level.

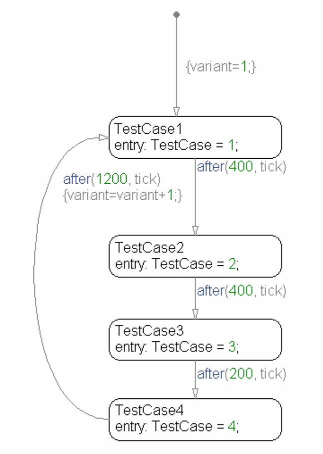


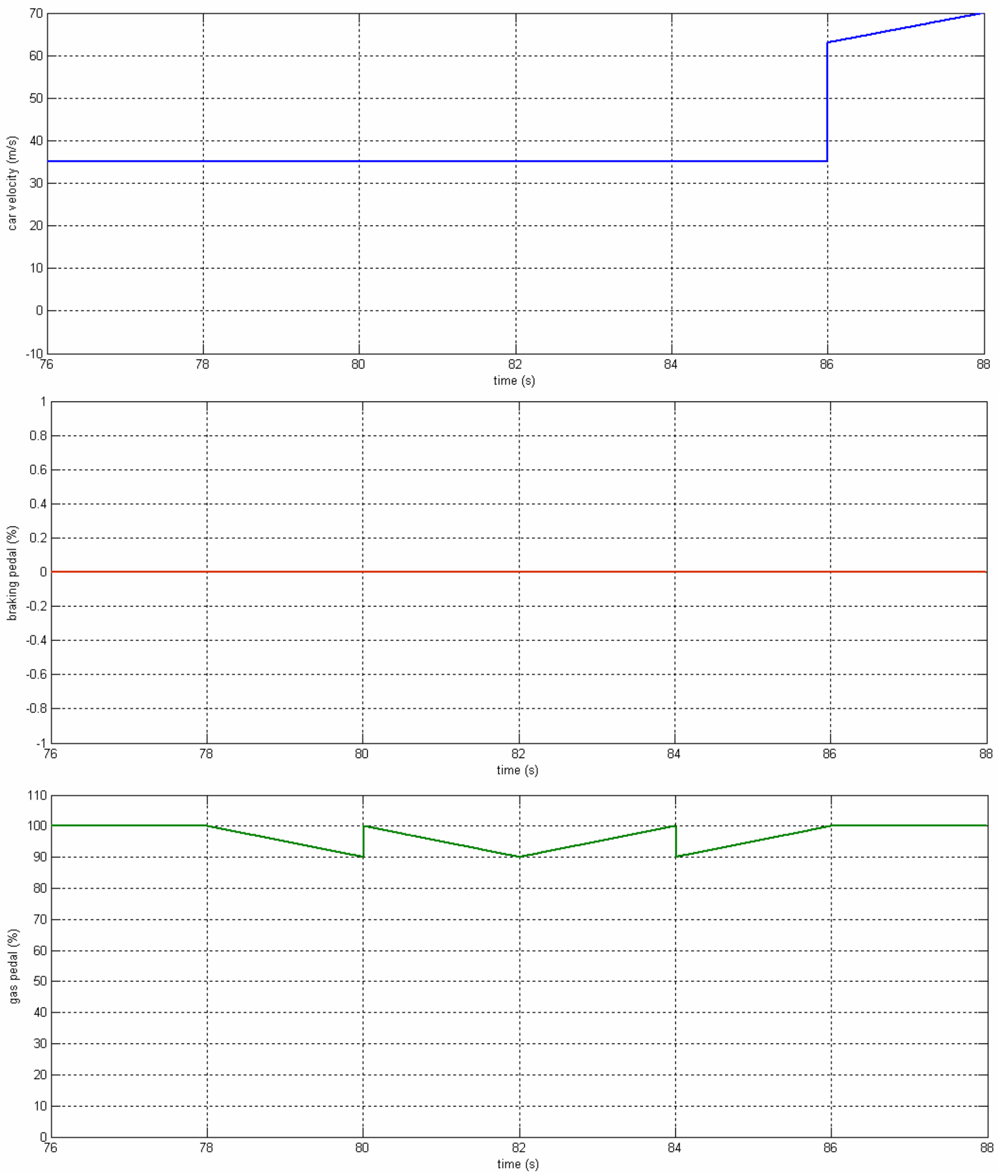
Figure 6.11: Test Control for Ordering the Test Cases Applying Minimal Combination Strategy.

6.2.5 Test Execution

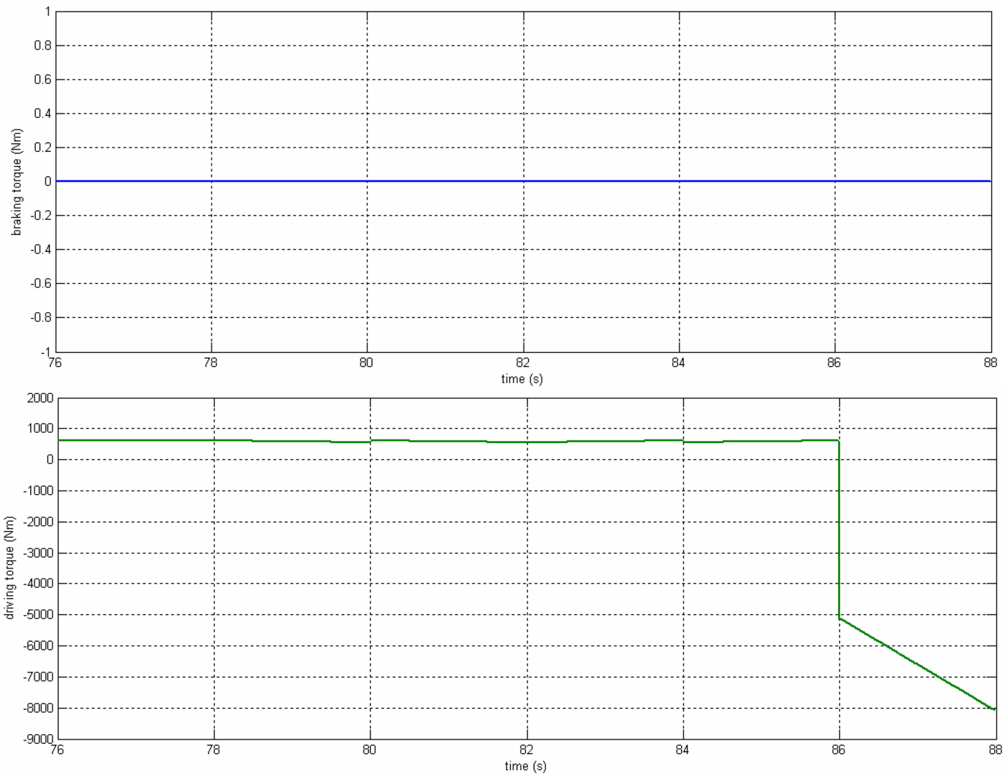
The test execution of the pedal interpretation case study is firstly discussed for a selected test case and then for the entire test suite. The test suite is repeated five times using different variant combinations.

In Figure 6.12, a selected combination of variants (numbered with 4) validating requirement 2.2 is shown. There, six test steps can be recognized. They are sequenced one after another so as to assert all VFs present within this requirement.

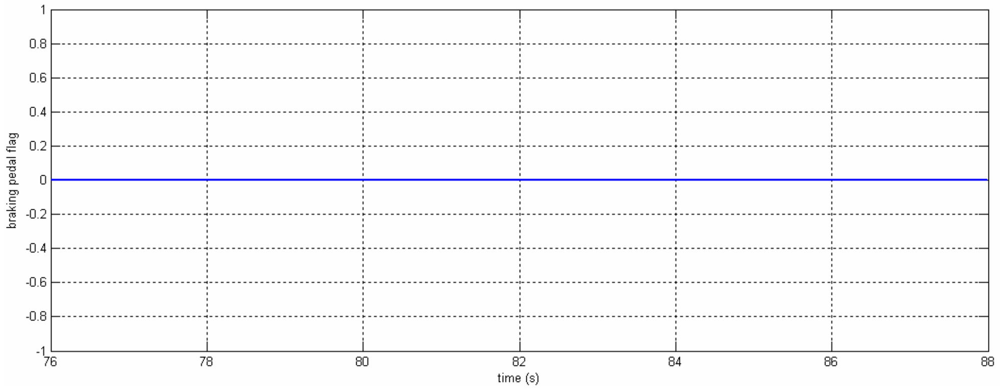
Now, the attention is focused to only one test step described in the previous section. Then, if the driving torque increases as expected, a *pass* verdict is delivered, otherwise a *fail* verdict appears. In Figure 6.12 a), the acceleration pedal (i.e., gas pedal) value increases in the time interval between 82 and 86 seconds and the velocity, if held constant. The local verdict (see Figure 6.12 d) drops down to *pass* for this particular situation, with a very short break for test steps switch (there, *none* verdict is monitored). This selected verdict provides the conclusion on only one single assertion. Every assertion has its own verdict. They are all summarized into an overall verdict simultaneously.

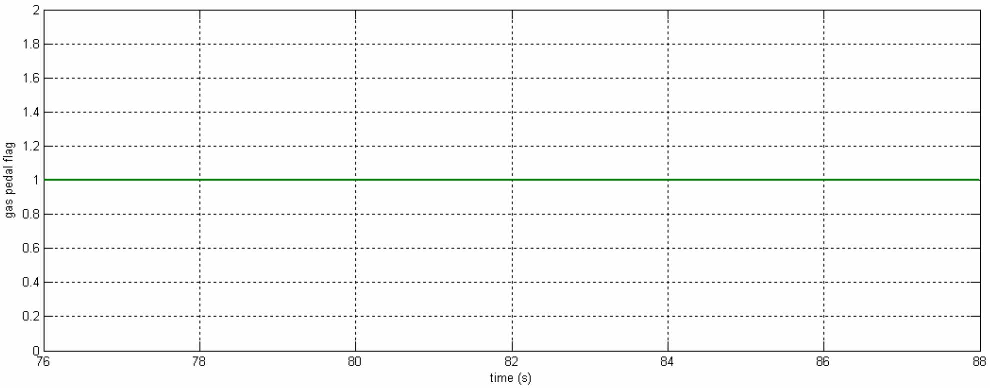


a) Applied Test Data Set

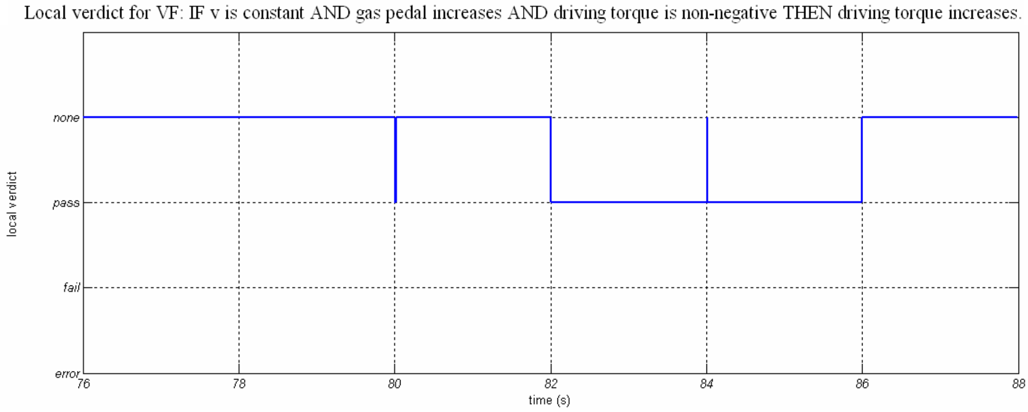


b) Obtained SUT Outputs





c) Obtained SUT Outputs



d) A Selected Local Verdict

Figure 6.12: Execution of Test Case 4 Applying the 4th Test Data Variants Combination.

Observing the SUT outputs (cf. Figure 6.12 b,c)), it is difficult to assess whether the SUT behavior is correct. Firstly, every single signal would need to be evaluated separately. Then, the manual process lasts longer then a corresponding automatic one and needs more effort. Also, the human eye is evidently not able to see all the changes. This already applies to the considered example, where the increase of driving torque is not easily observed, although it exists in reality. Further on, even if using the reference data so as to compare the SUT outputs with them automatically, it still relates to only one particular scenario, where a set of concrete test signals has been used. Regarding the fact that a considerable number of test data sets need to be applied for guaranteeing the safety of an SUT, it becomes evident and obvious how scalable the SigF-oriented evaluation process is and how many benefits it actually offers.

Figure 6.13 reflects the entire flow of signals produced for the pedal interpretation case study. Figure 6.14 and Figure 6.15 illustrate the SUT outputs just for orientation purposes. The test suite in the time interval between 0 and 22 seconds serves as a scheme for all the following suites, in which only the signals values vary, not the SigFs. All the SigFs may be traced back in Figure 6.8; therefore, no additional explanation on the flows will be provided here.

The local verdicts for assertions of each SigF can also be traced in time and are associated with the corresponding test steps (i.e., test cases and test suites).

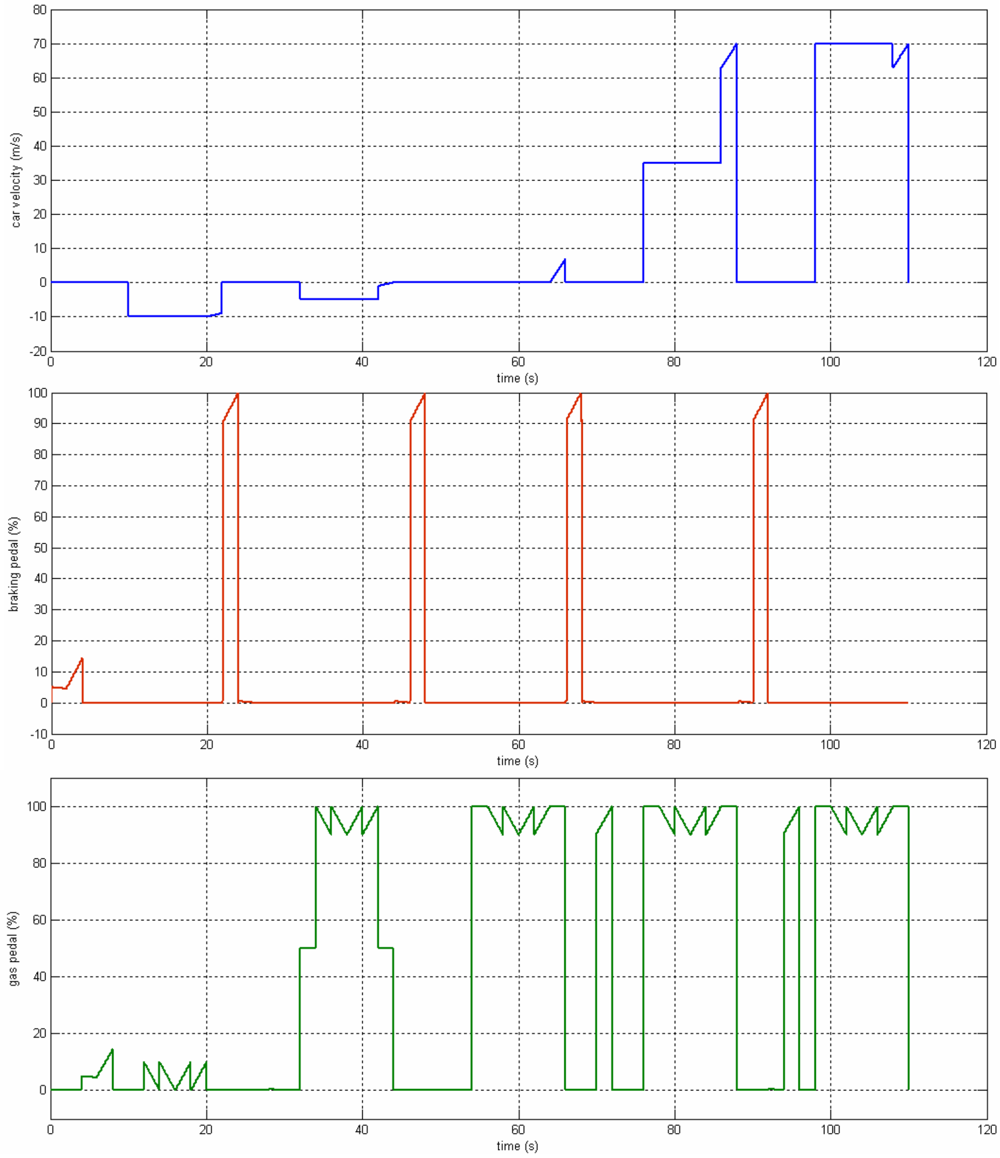


Figure 6.13: Resulting Test Data Constituting the Test Cases According to Minimal Combination Strategy.

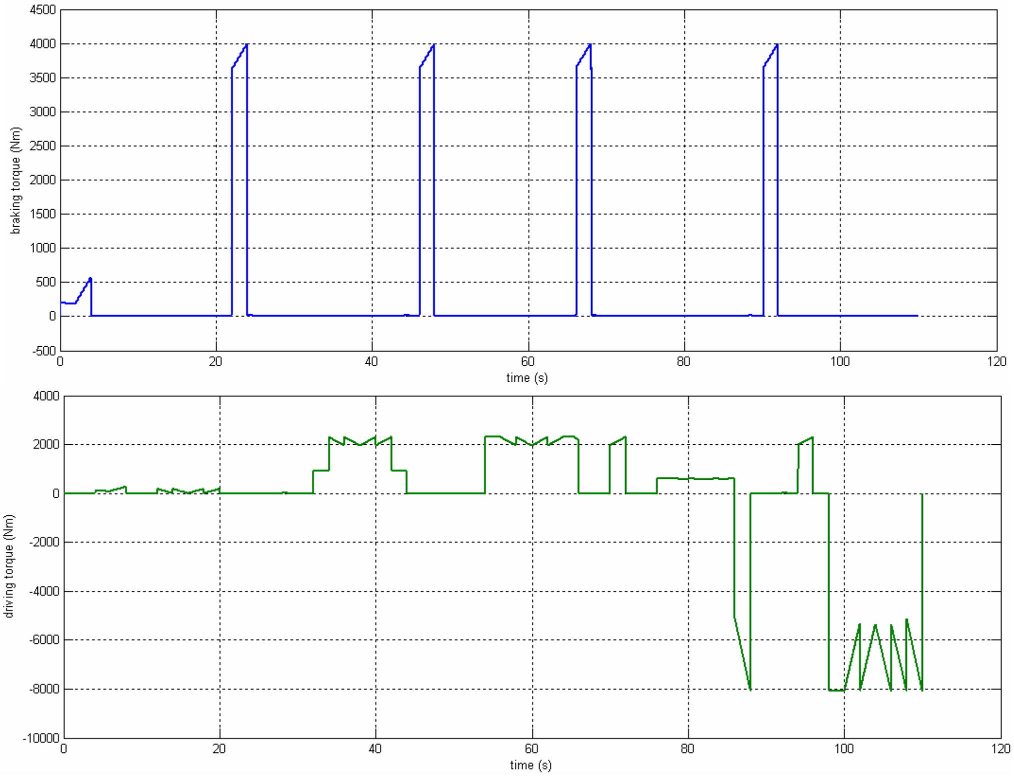


Figure 6.14: SUT Outputs for the Applied Test Data (1).

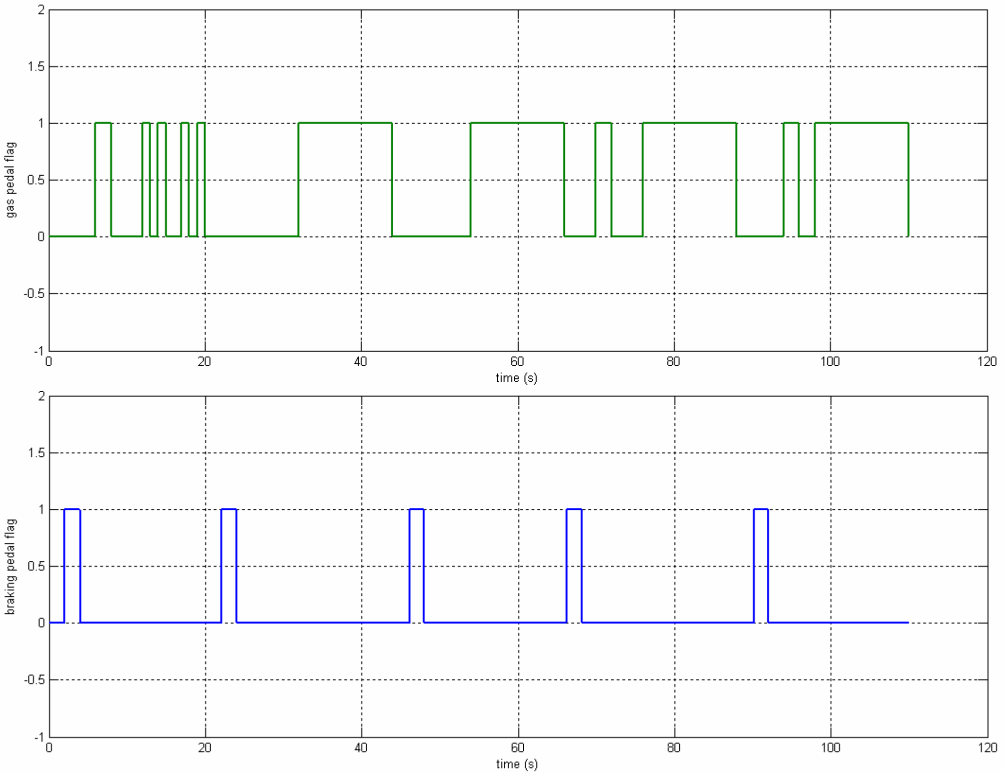


Figure 6.15: SUT Outputs for the Applied Test Data (2).

Let us consider the quality of the MiLEST approach. The execution of the tests using the generated test data in the pedal interpretation case study, let us recognize that several issues have not been designed in the VFs. While testing the SUT, some *fail* verdicts appeared. The reason for this was not an incorrect behavior of the SUT, but the imprecise design of the VFs with respect to the constrained ranges of data. Thus, the test specification has been refined (i.e., three VFs have been modified and two have been added). Then, the test data have been generated once again. The SUT issued a *pass* verdict at that point. This proves two facts: the TDG supports not only the test case production, but validates the defined test scenarios as well. By that, the test engineer is forced to refine either the SUT or the test specification, depending on the situation. The evaluation and validation of the proposed method will be discussed in Chapter 7 in detail.

6.3 Component in the Loop Level Test for Speed Controller

In this section, the tests of another ACC constituent are illustrated. This time, it is a component in loop with a vehicle model. Here, the test specification in the form of VFs is defined. Then, a particular attention is drawn to the test reactiveness so as to exemplify the concepts introduced in Section 5.5. The test reactiveness is exploited on the level of test data and test control.

6.3.1 Test Configuration and Test Harness

Figure 6.16 shows the speed controller as the SUT connected to a vehicle model.

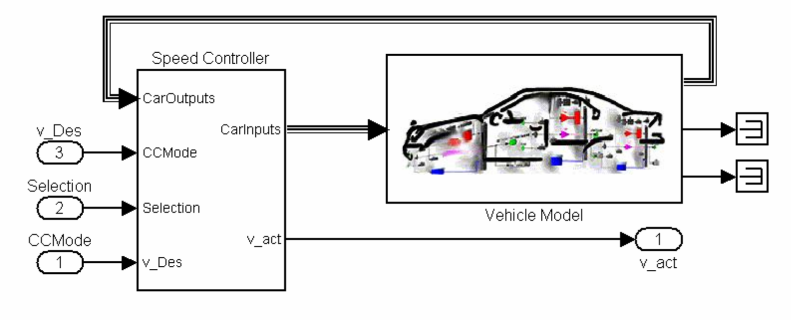


Figure 6.16: The Speed Controller Connected to a Vehicle Model via a Feedback Loop.

The SUT interfaces important for the upcoming considerations are listed in Table 6.5. The desired velocity (v_Des) at the SUT input influences the actual vehicle velocity (v_act) at the SUT output. Selection and CCMode are internal ACC signals that indicate the state of the whole ACC system. They will both be set to a predefined constant value to activate the speed control during its test. In this case study, v_Des and v_act play an important role. This practice simplifies the analysis of the proposed solution.

Table 6.5: SUT Inputs of Speed Controller.

Name of SUT signal	Desired velocity (v_Des)	Selection	Cruise controller mode (CCMode)	Vehicle velocity (v_act)
Values range	<10, 70>	{0, 1, 2, 3}	{0, 1, 2, 3}	<-10, 70>
Unit	m/s	–	–	m/s

In Figure 6.17, a test harness for the speed controller test is presented. Comparing it with the test harness for the pedal interpretation, the test control and its connections look different. The control is linked to both test specification and test data generation units. Hence, here the evaluation results influence the sequencing of test cases and by that the test stimuli activation algorithm.

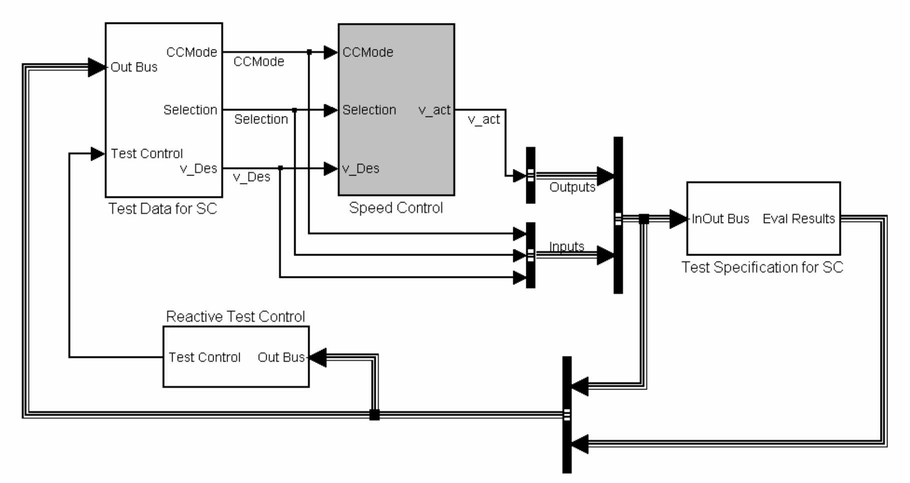


Figure 6.17: A Test Harness for the Speed Controller.

6.3.2 Test Specification Design

An excerpt of the functional requirements on the speed controller is given in Table 6.6. Very concrete requirements have been chosen so as to introduce the concepts more easily. Their realization at the test requirements level is shown in Figure 6.18.

Table 6.6: Requirements on Speed Controller.

ID	*Formal requirements on speed controller*
1	Speed control – step response: When the cruise control is active, there is no other vehicle ahead of the car, the speed controller is active and a complete step within the desired velocity has been detected, then the maximum overshoot should be less than 5 km/h and steady-state error should also be less than 5 km/h.
2	Speed control – velocity restoration time after simple acceleration/deceleration When the cruise control is active, there is no other vehicle ahead of the car, the speed controller is active and a simple speed increase/decrease applying a speed control button is set resulting in a 1-unit difference between the actual vehicle speed and the desired speed, then the speed controller shall restore the difference to <= 1 no later than 6 seconds after.

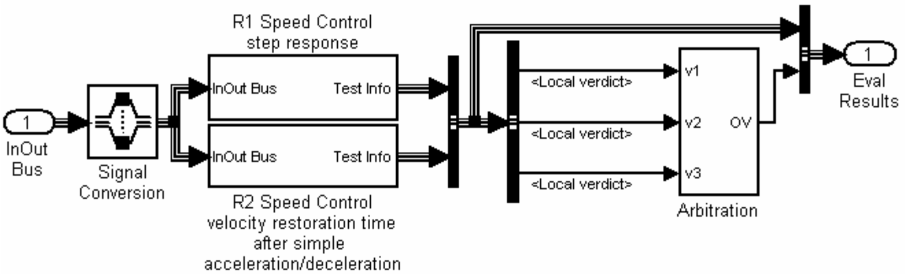


Figure 6.18: Test Requirements Level within the Test Specification Unit.

Concerning the first requirement, the steady-state error is checked. It permits for a deviation of ±5 km/h. Furthermore, the maximum overshoot of the vehicle velocity step response should not be higher than ±5 km/h.
In the second requirement, a comfortable speed increase/decrease of 1 km/h is being tested. Here, a button is used to set the speed change.
As a result of the requirements analysis, the following scenarios are obtained:

– IF Complete step within *v_Des* detected AND *CCMode*=active+no_target AND Speed Control *Selection*=active
THEN for *v_act* Maximum overshoot<5 AND Steady-state error<5.
– IF *CCMode*=active+no_target AND Speed Control *Selection*=active AND single *v_Des* step of size=⌉1| AND constant duration before the *v_Des* step=7 seconds
THEN after(6 seconds) |*v_Des*-*v_act*|<=1.

Then, the preconditions detect different sorts of steps within the desired velocity and they check whether the ACC is still active and no vehicle ahead appears, as presented in Figure 6.19.

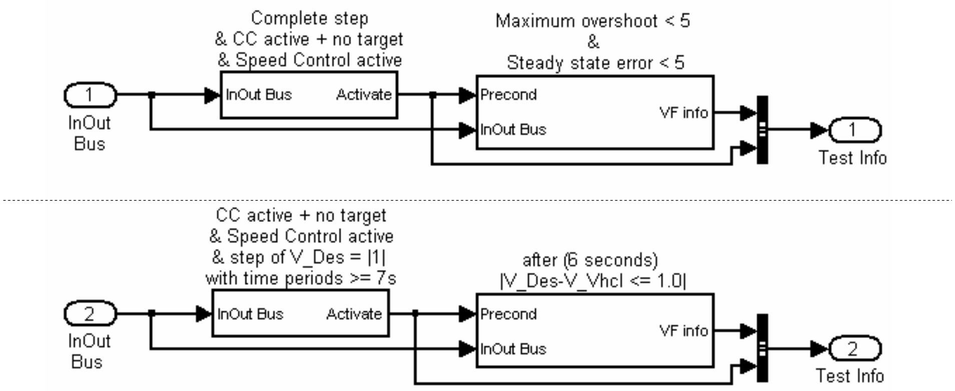


Figure 6.19: VFs within the Test Specification for the Speed Controller.

Preconditions and assertions pairs are implemented in the *Test Specification* part as two separate VFs each within a requirement. In the first VF the *Detect step response characteristics* pattern is applied to assess the test case. In the second VF a test pattern containing the temporal constraint (*after*) is used. Further details of their realization are omitted referring the reader to [MP07].

6.3.3 Test Data and Test Reactiveness

Figure 6.20 presents the abstract insights of the test data generation unit at the test requirements level that matches the test specification from Figure 6.18. For every requirement a subsystem is built and the appropriate signals are passed on to their switches. Depending on the test control conditions, a pre-selected test case is activated.

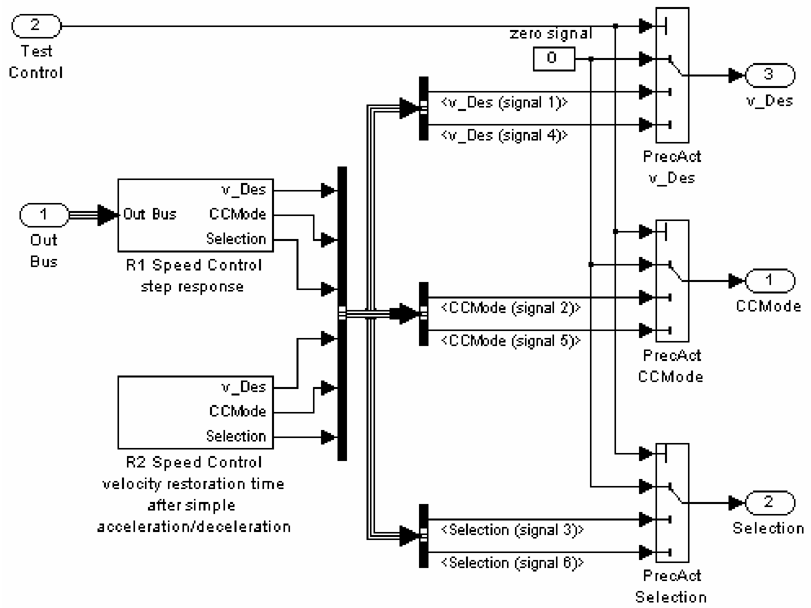


Figure 6.20: Test Requirements Level within the Test Data Generation Unit.

To validate requirement no. 1, *Selection* and *CCMode* are set to the predefined constant values that activate the speed control. Further on, at least two step functions are needed on the desired velocity signal. One of them should go up, the other down. Generators for such SigFs set are shown in Figure 6.21.

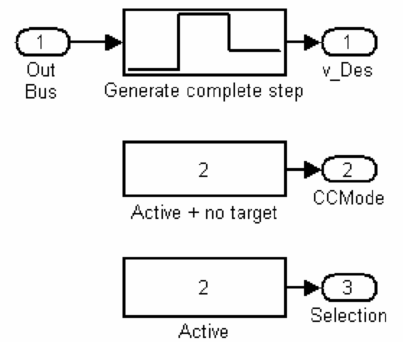


Figure 6.21: Test Data Set for Test Case 1.

It is difficult to establish when the next step function should appear as the stabilization time of the step response function is not known. Thus, the *evaluation trigger T* from the assertion of the corresponding VF (coming through the *OutBus* signal) is used to indicate the time point when the validation of the results from the previous step function has completed. In that case, the next step function can be started. This is realized within the *Generate complete step* subsystem as depicted in Figure 6.22.

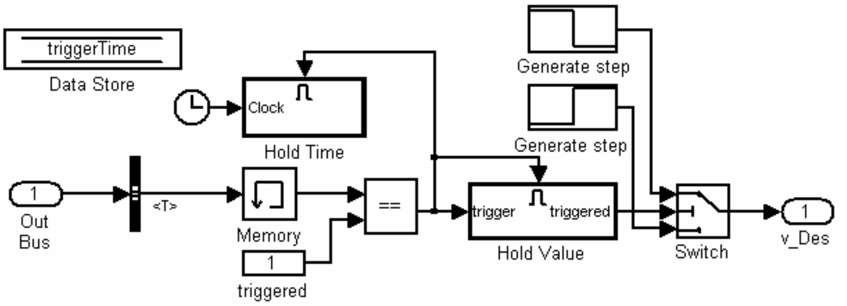


Figure 6.22: Influence of Trigger Value on the Behavior of Test Data.

Design principle no. 1 from Table 6.7 guarantees that if the first test data variant has been evaluated, the second one can subsequently be applied. Thus, the starting time of the second test data partition is established. The calculation is done automatically. The trigger *T* is checked. If it is activated, the time point is captured and the next step is generated. In this particular case a direct test reactiveness path is applied forwarding the signals from the VF directly to the test data unit without application of the test control unit.

In Table 6.7 the principles applicable for the speed controller case study are listed and the reference to the theoretical part initially provided in Section 5.5 is given.

Table 6.7: Design Principles Used to Support the Test Reactiveness in the Speed Controller Test.

no.	Context	Realization	Reference to theory
1	starting point of the next SigF variant generation within a single test case	IF *evaluation trigger=1* in the evaluation of a test case for an applied SigF variant THEN go on using the next *SigF variant*	1.3
2	test case duration	IF temporal constraint *x* determines a SigF under test, THEN test case *duration* is determined by a *SigF specific equation* (5.18)	2.3
3	starting point of the next test case	IF *verdict* of a test case equals to *pass or fail, or error*, THEN leave this *test case* at that time point and execute the next *test case*	2.4

Principle no. 2 allows for an automatic calculation of the test case duration. It is possible due to application of some specified criteria depending on the SigF under test. In this example the equation given in (5.18) applies, where:

- SigF specific time is the duration of a constant signal before the step and it equals at least 7 units,
- x equals at least 6.01 units following the time expression *'after (6, units)'* and the time step size of 0.01 units,
- i equals 1.

Hence, the calculated duration of *test case 2* is applied in the test control.

6.3.4 Test Control and Test Reactiveness

Finally, the test control specification for the speed controller is illustrated in Figure 6.23.

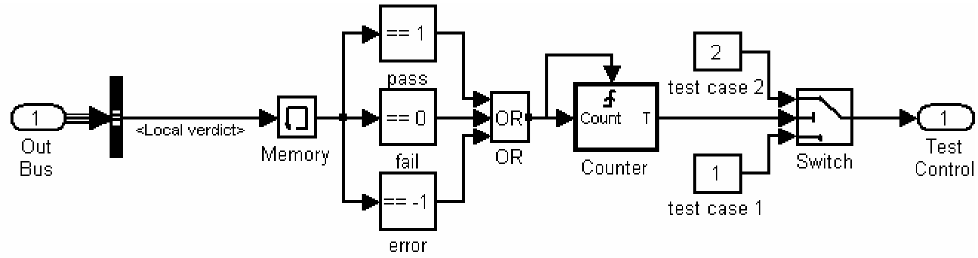


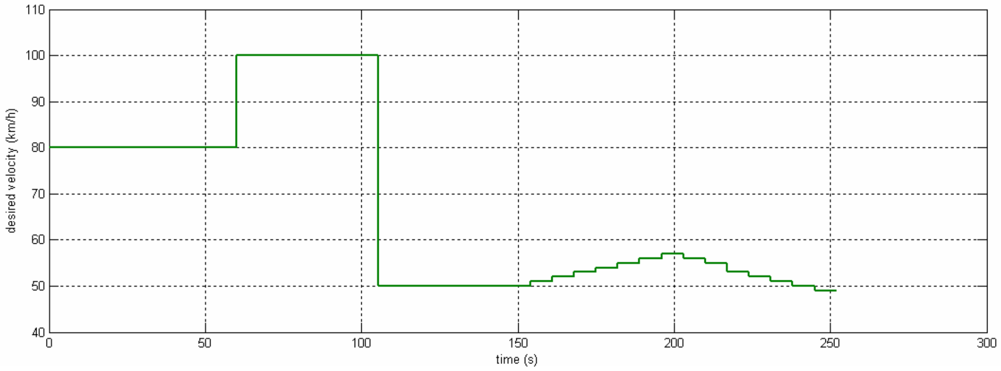
Figure 6.23: Influence of Verdict Value from Test Case 1 on the Test Control.

Here, *test case 1* validating requirement no. 1 is executed first. When the *local verdict* of *test case 1* appears to be different to *none*, the next *test case* is executed. There is no need to specify the duration of *test case 1* since the test control enables to redirect the test execution sequence to the next test case automatically whenever the evaluation of the previous test case has been completed. This is possible due to the application of principle no. 3 given from Table 6.7. The automation enables the test cases to be prioritized. Also, the testing time is saved. *Test case 2* is then executed.

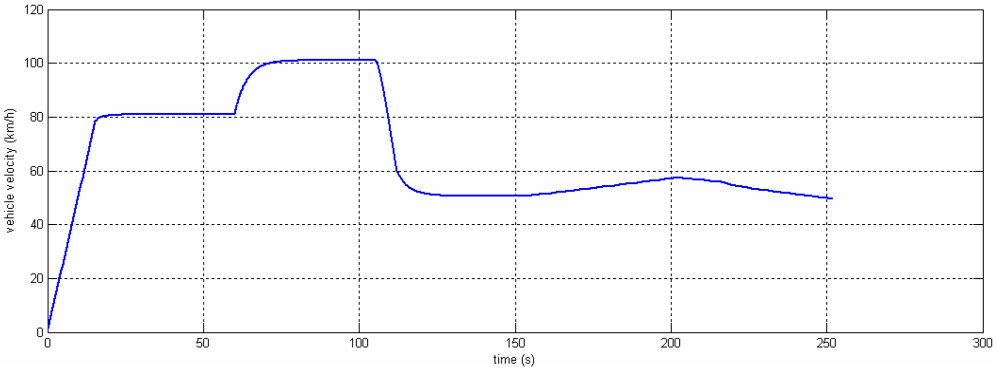
6.3.5 Test Execution

In Figure 6.24, the results of the test execution for the selected requirements of the speed controller are drawn. Firstly, in Figure 6.24a the desired velocity (provided in km/h) is constant, then it increases in time; hence the vehicle velocity in Figure 6.24b increases as well. When the step response within the vehicle velocity stabilizes, the test system assigns a verdict to the appropriate test case (as illustrated Figure 6.24c and d) and the next test data set is applied. The desired velocity decreases; thus, a decreasing step response is observed and a verdict is set. Further on, the next test case is activated. The stairs function is applied and appropriate verdicts are assigned. The stairs stimulate the SUT multiple times following the scenario when a car driver accelerates/decelerates considerably by pressing a button.

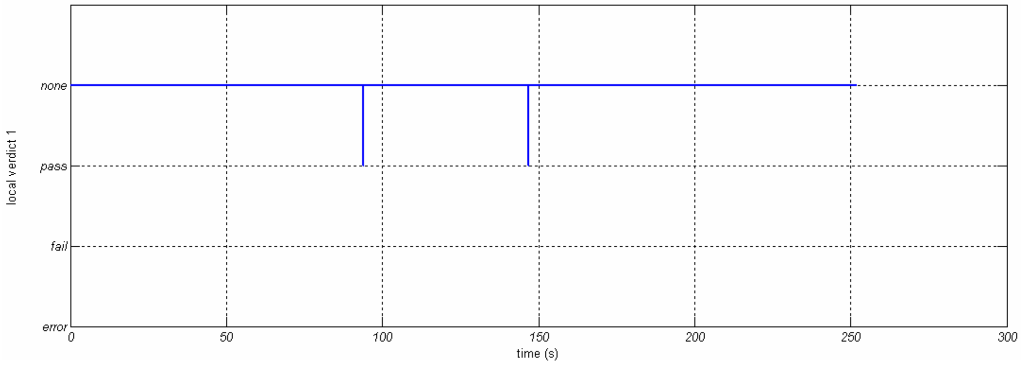
Summing up, two test cases containing sequences of different test data variants are executed. The local verdict values are shown in Figure 6.24c and d. Only *none* and *pass* appear which indicates that the SUT satisfies the requirements. No faults are found (i.e., no *fail* verdicts appear).



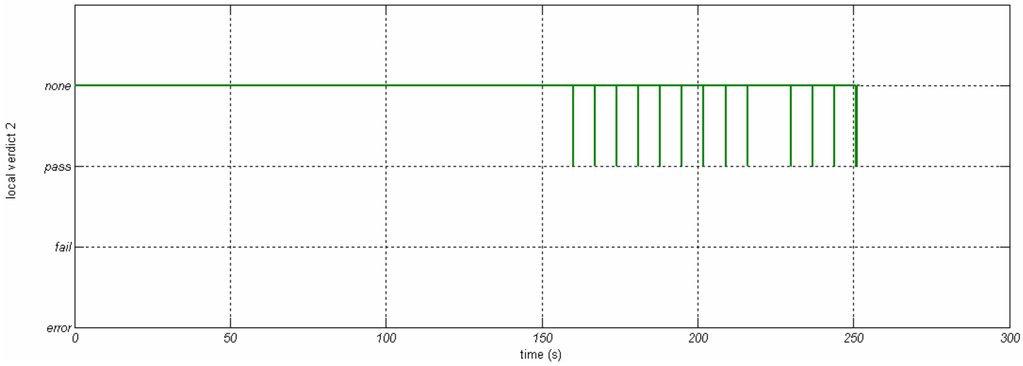
a) Applied Test Data



b) Obtained SUT Output



c) Local Verdict 1



d) Local Verdict 2

Figure 6.24: Results from the Test Execution of the Speed Controller.

6.4 Adaptive Cruise Control at the Model Integration Level

In the following section, the third case study is presented. It is the ACC itself used for representing the model integration level testing. The general requirements for the ACC have already been introduced in Section 6.1. Some of them are recalled now in order to illustrate the integration test in a relatively simple manner. In this example, besides the VFs, specific interaction models are provided. Also, the concept of a test sequence is applied and put in relation with the defined test control specification.

6.4.1 Test Configuration and Test Harness

The test harness for the ACC is shown in Figure 6.25. In contrast to the previous case studies an additional bus creator called *Intermediate* appears. It collects the signals present between different components within the system. These are needed to be able test whether the integration of these units is working properly.

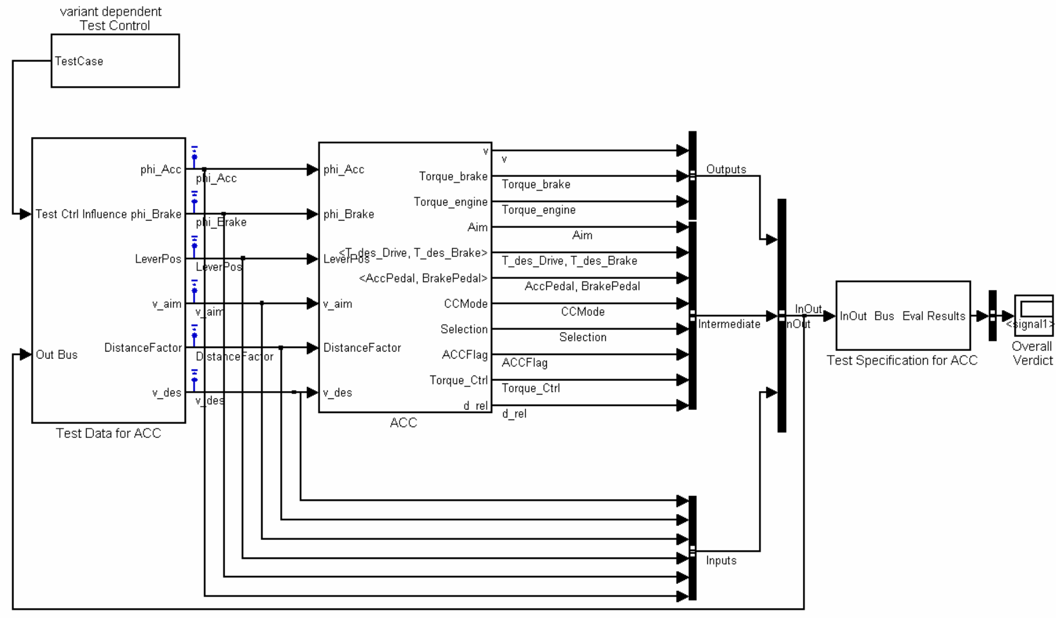


Figure 6.25: Test Harness for ACC.

6.4.2 Test Specification Applying Interaction Models

ACC behavior can be divided into a few services interacting with each other. A general ACC flow scenario is presented in Figure 6.26 applying the *High level hybrid Sequence Charts for testing (HhySCt)* notation that has been described in Section 5.6. Here, the ACC system must be activated first. This activation enables the velocity or distance controller to be started (*Adjust-Distance/Velocity*). The ACC may operate in two modes – it can either adjust the distance or the velocity. The system may be deactivated by braking coming out from two services: ACC being active (*ActivateACC*) or ACC already controlling the velocity of a car (*AdjustDis-*

tance/Velocity). The controlling activity happens continuously. The system may be switched off temporarily when the car driver accelerates. When the acceleration activity stops the automatic reactivation of the ACC system appears and it starts to work in the control mode again.

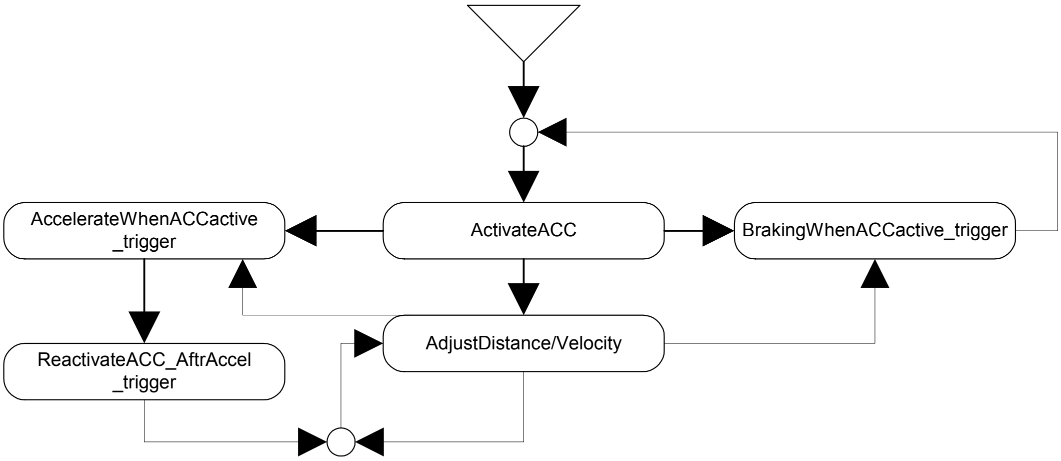

AccelerateWhenACCactive _trigger
ActivateACC
BrakingWhenACCactive_trigger
AdjustDistance/Velocity
ReactivateACC_AftrAccel _trigger

Figure 6.26: ACC Flow – Possible Interactions between Services.

Adjusting the velocity or distance may happen only after the activation of ACC. Then, the braking service (shown in Figure 6.27) comes into play. It is designed using *hySCt*. The assumptions resulting from the previous activities are that:

- the car is in a movement,
- the desired speed is set on a predefined speed level such that it is higher than 11 m/s,
- and the ACC is switched on.

If this is the case and the *braking pedal* is pressed (*phi_Brake>ped_min*), then the *pedal interpretation* unit sets the braking flag on (PedalBrake=1); the *braking torque* is adjusted according to the formula *T_max_Brake/T_des_Brake=phi_Brake/phi_Max*. Additionally, the ACC flag is set on non-active (*ACCFlag=0*) and the cruise controller mode is set on non-active (*CCMode=0*). As a result the car decelerates (*v decreases*). With this practice, the ACC is switched off. The activities of all the specified units should happen simultaneously.

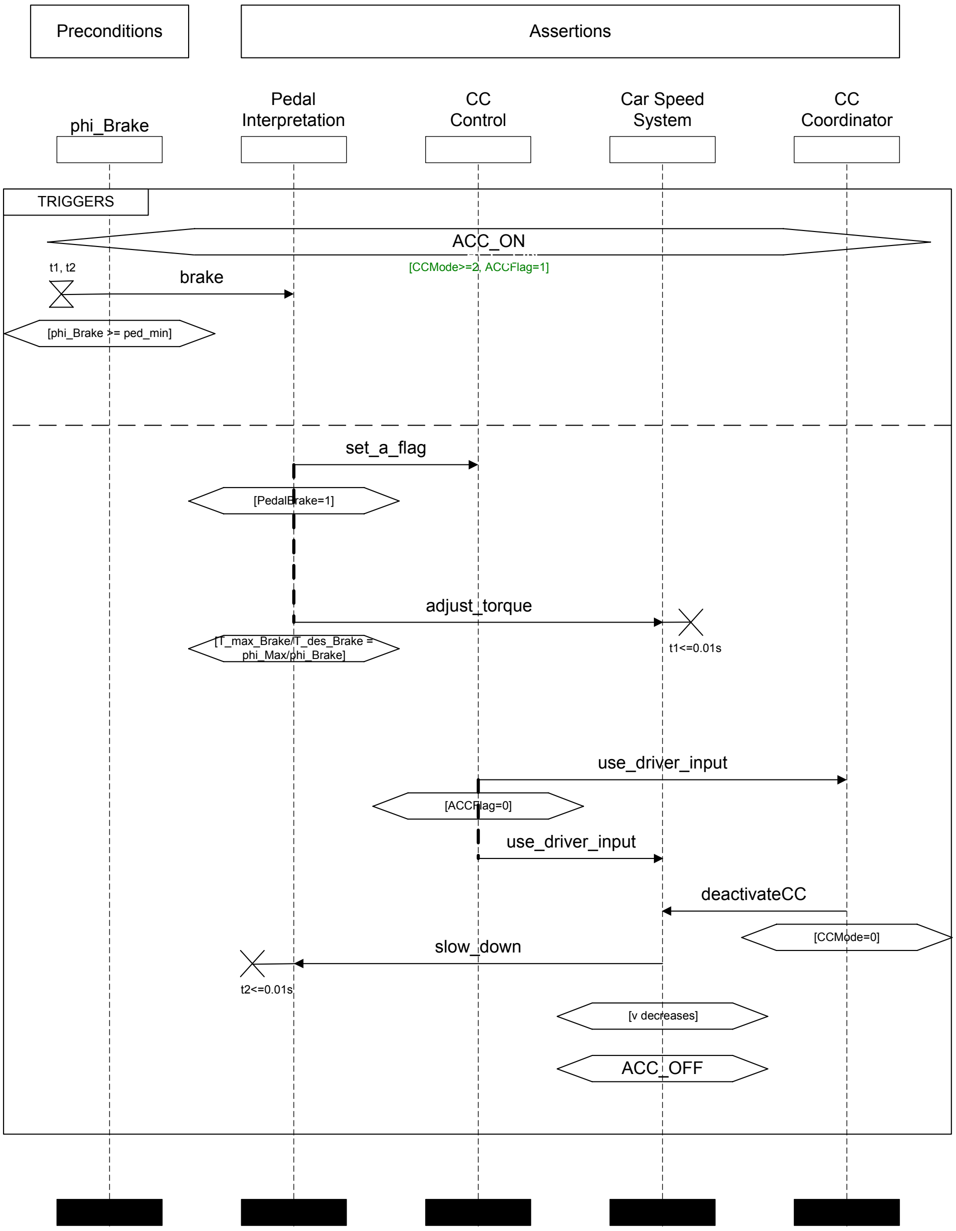


Figure 6.27: Service: Braking when ACC System is Active Trigger.

6.4.3 Test Specification Design

The implementation of the test specification in MiLEST is generated from the H*hySCts* and *hySCts* provided in the previous section. An example set of the VFs is given in Figure 6.28. Some of them relate 1-to-1 to the *hySCts*, whereas the others focus on selected aspects (e.g., timing issues) of the services.

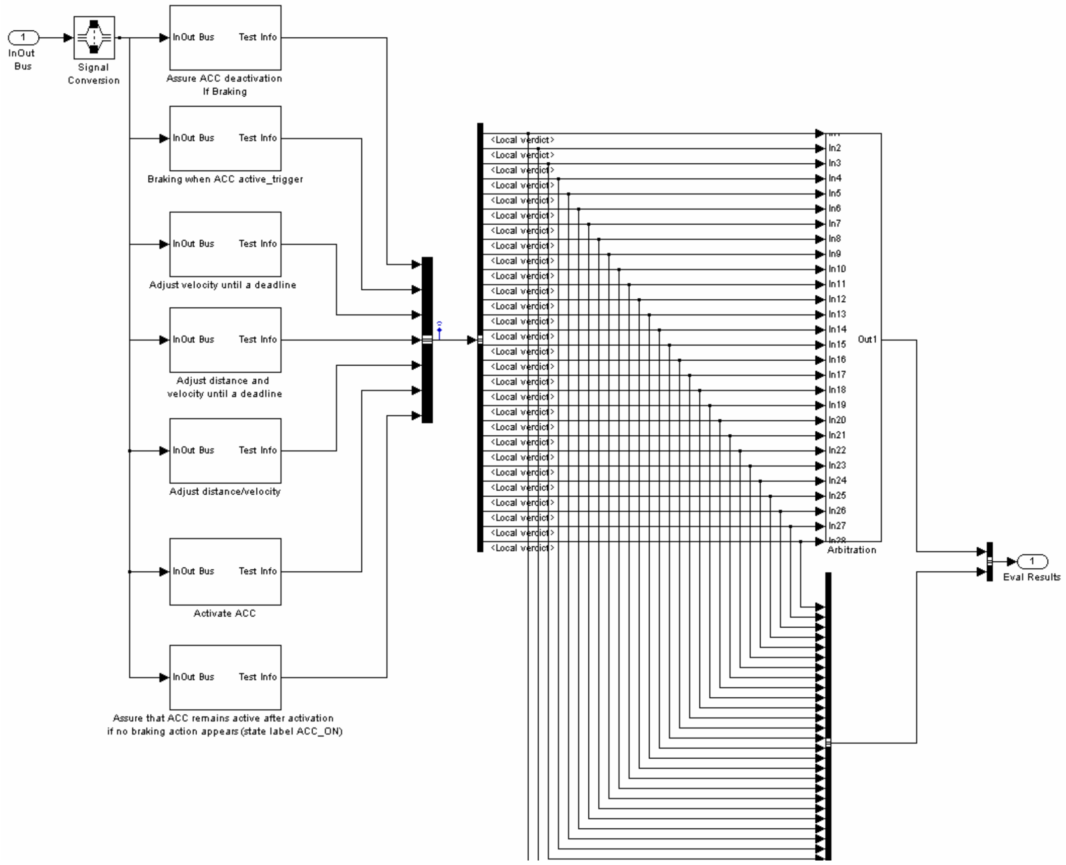

1
InOut Bus
Signal Conversion
InOut Bus
Test Info
Assure ACC deactivation If Braking
Braking when ACC active_trigger
Adjust velocity until a deadline
Adjust distance and velocity until a deadline
Adjust distance/velocity
Activate ACC
Assure that ACC remains active after activation if no braking action appears (state label ACC_ON)
<Local verdict>
In2
In3
In4
In5
In6
In7
In8
In9
In10
In11
In12
In13
In14
In15
In16
In17
In18
In19
In20
In21
In22
In23
In24
In25
In26
In27
Out1
Arbitration
1
Eval Results

Figure 6.28: A Set of VFs Retrieved from the hySCts.

In the following only one VF is presented in detail for the simplicity reason.
It is called *Braking when ACC active_trigger* and its implementation is proposed in Figure 6.29 and in Figure 6.30. In the *preconditions* part (see Figure 6.29), the braking pedal signal is monitored. If the pedal is pressed and the gas pedal is not pressed simultaneously it is checked whether the ACC has been active before. If this is the case, the *preconditions* activate the *assertions*. Within the *assertions* (Figure 6.30) several SigFs are tested. If *CCMode* and *ACCFlag* indicate that the ACC system has been switched off, the *velocity* decreases, the *braking torque* behaves according to a predefined formula and the *pedal flag* is true, then the test provides a pass verdict. Otherwise a failure is detected.

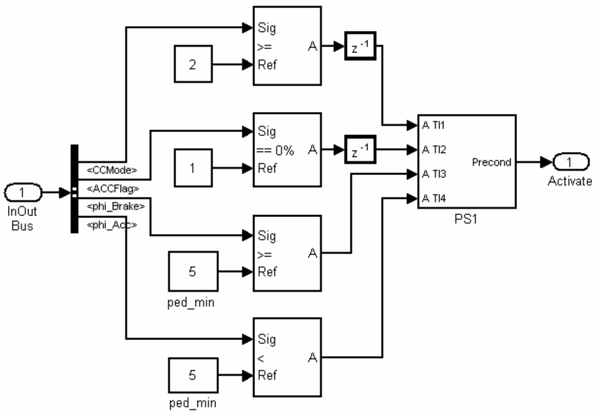


Figure 6.29: Preconditions for the VF: Braking when ACC active_trigger.

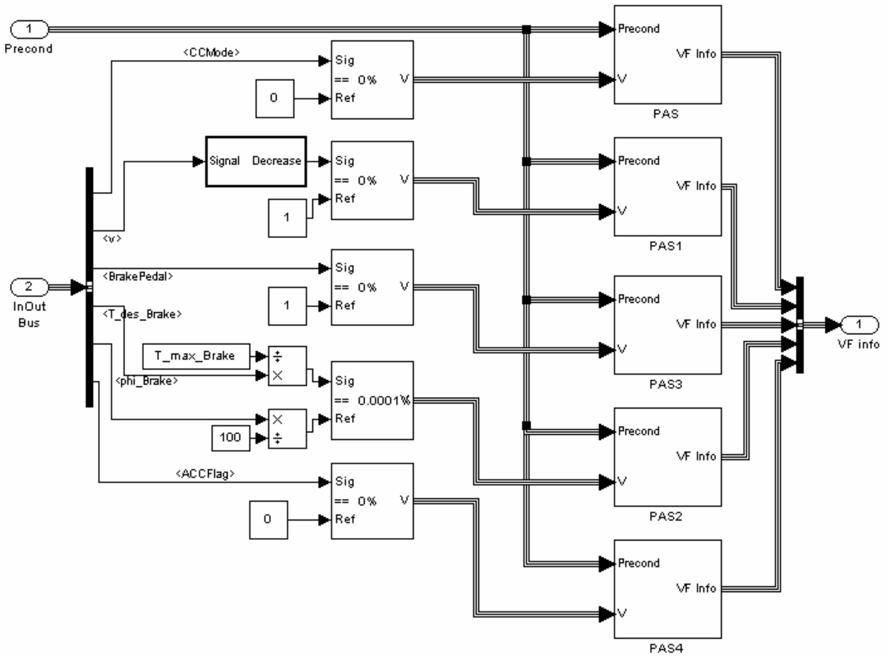


Figure 6.30: Assertions for the VF: Braking when ACC active_trigger.

6.4.4 Test Data Derivation

The structural framework managing the created test data is created automatically as a result of the transformation from the given VFs. It is presented in Figure 6.31. Here, only three data sets are obtained since only the left-hand flow from Figure 6.26 is analyzed in this case study. The restriction cuts the search space to the path: *Activate ACC → Adjust Velocity → Braking when ACC active*. Every set is responsible for one test case. Only the entire set of interactions forming the services is considered during the transformation. The remaining test objectives are activated implicitly by these sets of data.

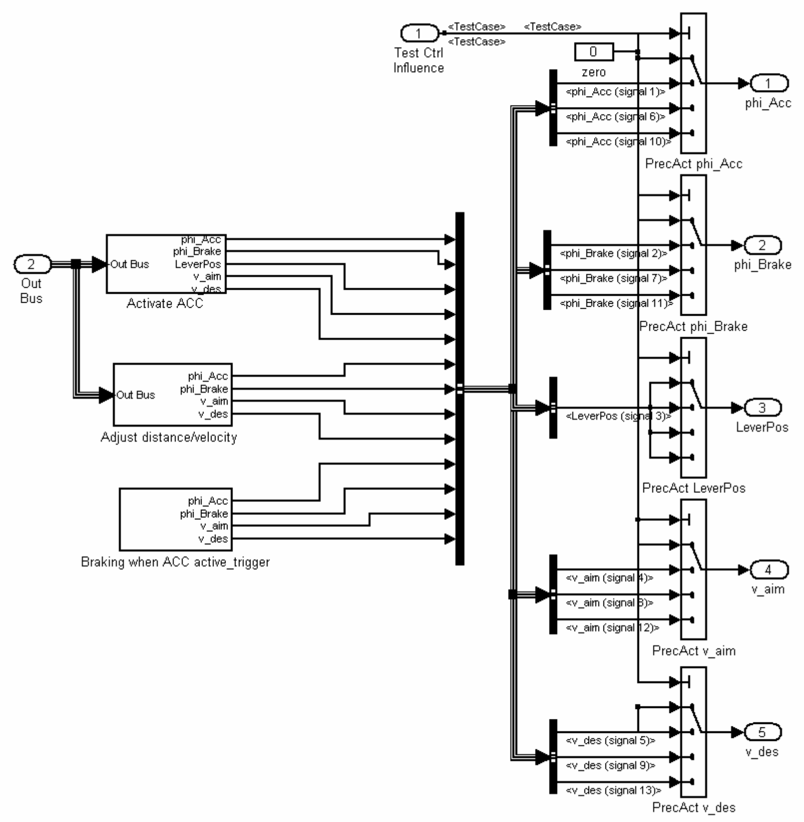
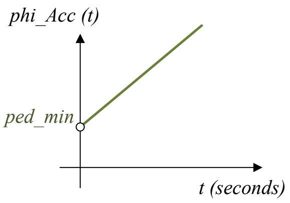
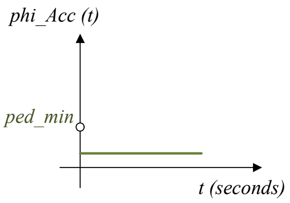
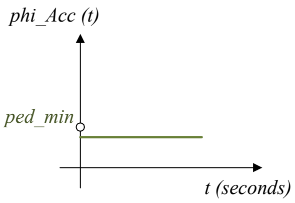
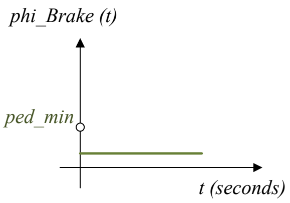
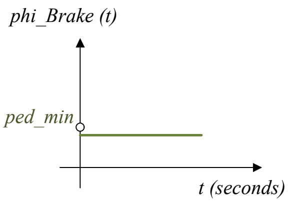
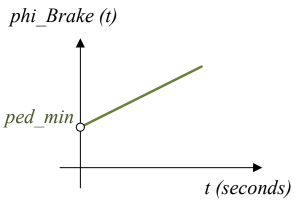
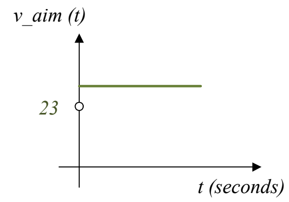
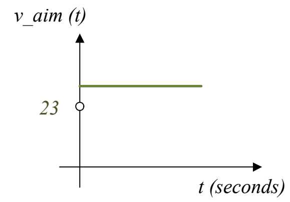
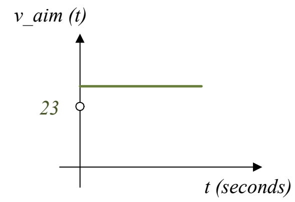
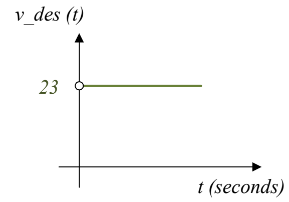
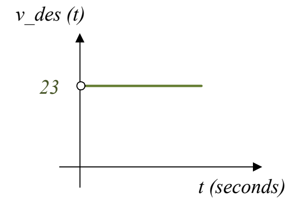
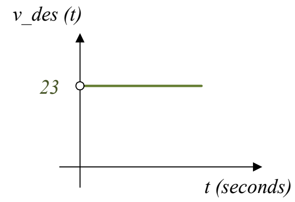
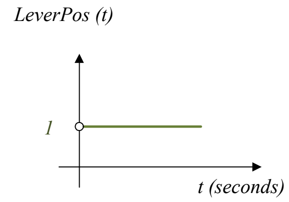
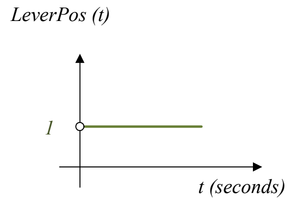
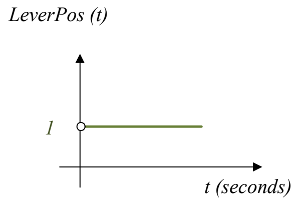


Figure 6.31: Test Data Set for a Selected ACC Model Integration Test.

The contents of the generated test data sets are limited too. They are provided in the form of parameterized generators of SigFs. The test data variants are generated automatically based on the boundaries and partitions analysis, similar as for the previous case studies. In Table 6.8 only an abstract overview of a selected set of the produced data is given for reasons of simplicity. The main issue is to understand the reasons for generating the SigFs and the meaningful constraints put on the ranges of those signals. Hence, the analysis leading to the concrete representatives is omitted here.

Table 6.8: Relation of Test Data Sets to Services.

Services **Interfaces**	**ActivateACC**	**AdjustDistance/Velocity**	**BrakingWhenACCactive_trigger**
phi_Acc			
phi_Brake			
v_aim *related to v_des value*			
v_des *constant throughout the entire test sequence*			
LeverPos *activated throughout the entire test sequence*			

Additionally, Table 6.8 includes the sets of the created test data in relation to the previously specified services. The SigF generators are derived directly from the VFs; however, indirectly they are based on the detailed specification of services. This indirect relation results from the fact that the VFs themselves are the targets of the transformation from the services descriptions.

Then, also at this level the information gained from the *HhySCts* is leveraged. The sequence of services provides the knowledge, how some SigFs should behave in combination with the other SigFs so as to assure the proper flow of the entire test scenario.

Here, the concept of the so-called *test sequence* is applied. The test sequence for the considered flow (cf. Figure 6.31) is exactly the same as the content of the test control (see Figure 6.32). It happens because only one path of the *HhySCt* has been exploited. If all the paths are executed, though, the test control includes more test sequences. Their number is equal to the number of paths resulting from the path coverage criterion.

6.4.5 Test Control

Sequencing the test data sets on the higher abstraction level (i.e., here in the test control part in Figure 6.32) is derived from the *HhySCts* too. In that case, it is a one-to-one mapping of the left-hand flow from Figure 6.26 since the analysis of the case study is restricted to only one path. Eventually, the timing issues are added so as to make the test model executable according to the algorithm provided in Section 5.4.4. The test control design is given in Figure 6.32.

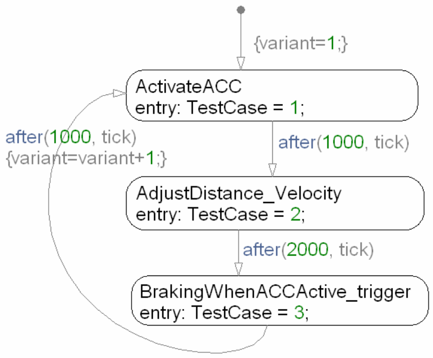


Figure 6.32: Test Control for a Selected ACC Model Integration Test.

6.4.6 Test Execution

While executing the tests of this case study, the first phase is the activation of ACC. Further on, the velocity adjustment occurs and then braking appears. By that, three test cases arranged in one test suite are obtained. Every combination of variants appears in the same sequence restricted by the test suite within the test control.

Going through the results of the test execution, the function of the car velocity in time is obtained. As one can observe the velocity increases in the first 20 s of the simulation (as specified in the first column of Table 6.8 and caused by pressing the gas pedal). Further on, the velocity is pending so as to achieve 23 m/s as the ACC is already activated. Finally, after 60 s, braking activity appears. At 80 s the test suite finishes. Additionally, the next variants set of signals combination is applied. The behavior corresponds to the previous one, although a slightly different shape of the car velocity is observed.

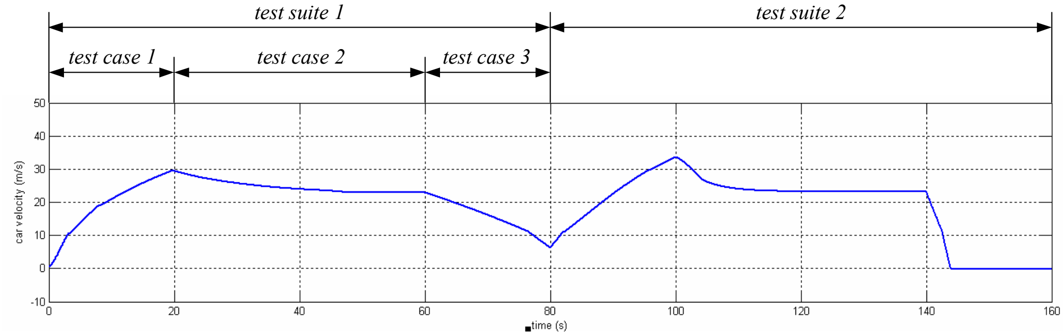


Figure 6.33: The Car Velocity Being Adjusted.

In these test scenarios all the test cases pass. So, the SUT behavior is the same as expected. Nevertheless, further variants of the test stimuli should be applied to complete the test coverage for this case study. This need is indicated by the test quality metrics values that will be calculated in Section 7.2.

6.5 Summary

The case studies are a means to show the feasibility of MiLEST. Three of them have been performed; all of them are related to the ACC system design.

The functional requirements and the model of ACC have been provided in Section 6.1. Then, the pedal interpretation and the speed controller have been extracted from it. The first example has been analyzed in Section 6.2 representing a test at a component level. Here, the test specification and data generation algorithms have been described in detail. The next example has been discussed in Section 6.3 and the concepts of a test at the component in the loop level have been reviewed. Also, reactive testing has been illustrated in this section.

Finally, in Section 6.4, the ACC itself as an SUT for model integration level testing has been investigated. The *HhySCt* and *hySCt* models have been designed so as to define the functional relations between the interacting requirements for testing purpose. The test data have been retrieved by assuring the coverage of the selected services specified in the *HhySCt*.

All the tests have been executed returning the verdicts.

The concepts of completeness, consistency and correctness of the designed test models for every single case study will be discussed in Section 7.3. Also, the executed test cases will be carefully evaluated there.

7 Validation and Evaluation

"One never notices what has been done;
one can only see what remains to be done."

- Maria Skłodowska – Curie

The following chapter investigates *quality* aspects of the test approach proposed in this thesis. In particular, *test metrics* are defined so as to measure and, by that, assure the consistency and correctness of the proposed test method. The *tests* can reveal high *coverages* with respect to *different test aspects* only if the corresponding test metrics provide a proper level.
Besides, in this chapter, the test strategy is considered. Obviously, the prototype is provided so as to validate the concepts developed in this thesis.

Section 7.1 outlines the details of the realization of Model-in-the-Loop for Embedded System Test (MiLEST) attached to this thesis. The components of its implementation are additionally listed in Appendix E. In Section 7.2, the quality of the test specification process, the test model, and the resulting test cases are investigated in depth. Firstly, the test quality criteria are reviewed. Then, the test quality metrics are defined and explained. Finally, they are classified, summarized and compared with related work.
Section 7.3 presents test quality metrics calculated for the test models of the case studies given in Chapter 6. Further on, in Section 7.4, the applied test strategy is contrasted with other approaches, commonly known in the automotive domain. At the end, in Section 7.5, scope and limitations of MiLEST are outlined. Finally, a short summary in Section 7.6 concludes this chapter.

7.1 Prototypical Realization

MiLEST is a Simulink (SL) add-on built on top of the MATLAB (ML) engine. It represents an extension towards model-based testing activities as shown in Figure 7.1.

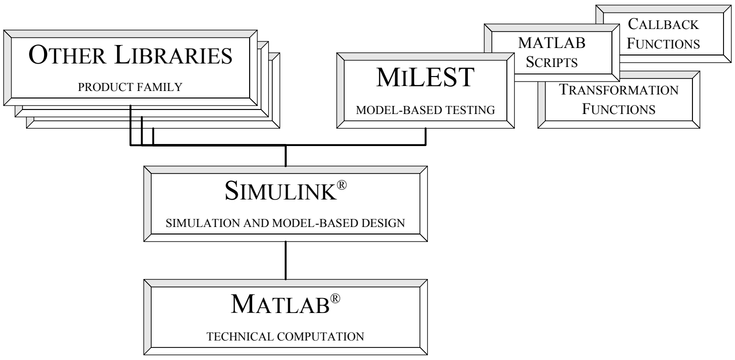


Figure 7.1: Integration of MiLEST in the MATLAB/Simulink framework.

MiLEST consists of an SL library including callback functions, transformation functions, and other ML scripts. The testing library is divided into four different parts as illustrated in Figure 7.2. Three of them cover the units present at the test harness level. These are: test specification, test data, and test control. Additionally, the test quality part includes elements for assessing the quality of a given instance of a test model by applying the metrics.

The callback functions are ML expressions that execute when a block diagram or a block is acted upon in a particular way [MathSL]. These are used together with the corresponding test patterns to set their parameters.
Additionally, transformation functions and quality metrics have been realized. Their application is described in Appendix E. A simple example of an ML function call is the main transformation where two input parameters must be entered by the test engineer manually so as to indicate the names of the system under test (SUT) and the resulting test model as presented in (7.1).

```
TestDataGen('Pedal_Interpretation_test', 'PedalInterpretation')     (7.1)
```

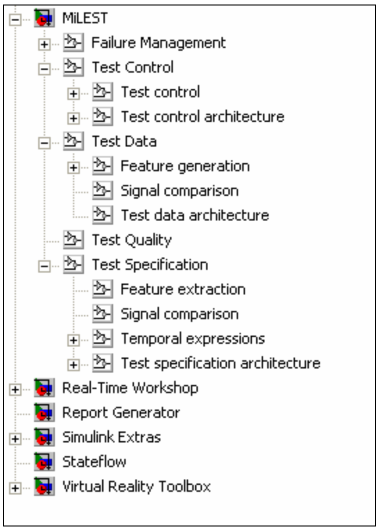


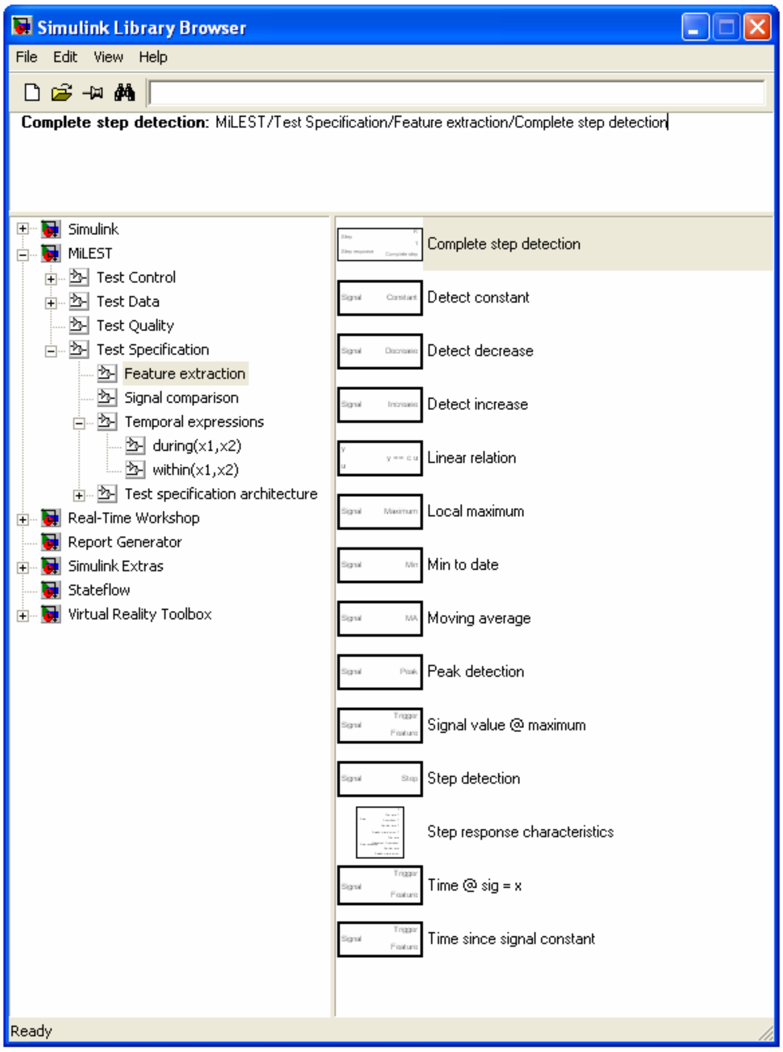
Figure 7.2: Overview of the MiLEST Library.

Different test activities are divided according to a scheme. In each case a hierarchical architecture is provided and separately the elements of the corresponding units are collected. These patterns correspond to the levels discussed in Section 5.1, providing different abstractions for different functionalities.

Taking the patterns of the test specification (TSpec) as an example, a Simulink (SL) subsystem called <Test specification> is present in <Test specification architecture> tag. Here, the construction of validation functions (VFs) is possible. Also, the signal evaluation functionality is available, including both feature extractors under <Feature extraction> and signal comparison blocks under <Signal comparison>. <Temporal expressions> are helpers for designing a full TSpec as discussed in Section 4.1.4 and Section 5.2.

By default, the <Test specification> contains a single requirement pattern with both a preconditions and assertions block, each of which includes one single precondition and one single assertion. If further requirements need to be inserted, the callback function for the subsystem enables it to be parameterized. Then, more requirements can be added into the design (cf. Figure 4.39 in Section 4.4). Applying this practice down to the feature detection level guarantees that large test system structures may be generated quickly and efficiently. In the MiLEST library the arbitration algorithm, local verdicts bus and the verdict output are additionally included so as to use the TSpec as a test evaluation mechanism.

At the feature detection level, more manipulation is needed since here the real content of the TSpec is provided. The feature extraction and the signal comparison blocks differ, depending on the functionality to be validated. Both <Feature extraction> and <Signal comparison> blocks need to be replaced in the concrete test model by using the instances from the library (cf. Figure 4.40 in Section 4.4, Figure 5.2 and Figure 5.3 in Section 5.2). The implemented entries of <Feature extraction> are listed in Figure 7.3.



Complete step detection: MiLEST/Test Specification/Feature extraction/Complete step detection

Figure 7.3: Overview of MiLEST Library.

In the pattern for preconditions specification (cf. Figure 4.39 in Section 4.4) a single PS block is available. It is parameterizable and synchronizes different signal features (SigFs). In the parameter mask the identification delays of the triggered and non-triggered SigFs can be entered. The pattern for assertions specification includes a PAS block instead. In contrast to the previous case a single PAS block is needed for every extracted feature. The PAS block can be configured according to the SigF type being asserted, changing w.r.t. the number of inputs.

Similar procedures apply to the <Test data generator> embedded in the <Test data architecture>. However, the guidelines are only needed when it is created manually. Since it is produced automatically in the proposed approach, the details will not be described here.

Also, the test control can be constructed automatically. However, a number of elements to be added manually may enrich its logic. These are: <Test control conditions> and <Connection

helpers>. They include algorithms that restrict the test case execution based on the verdicts, test stimuli values or test evaluation information at a certain time point or in given time interval.

A good practice is to rename all the applied patterns according to the concrete functionality that they contain.

7.2 Quality of the Test Specification Process and Test Model

7.2.1 Test Quality Criteria

As far as any testing approach is considered, *test quality* is given a lot of attention. It constitutes a measure for the test completeness and can be assessed on different levels, according to different criteria. In the upcoming paragraphs three main categories of the test quality resulting from the analysis of several efforts in the related work are distinguished. All of them are based on the functional considerations, leaving the structural[41] issues out of the scope in this thesis.

Primarily, criteria similar to that for software development are of importance. Hence, the same as the consistency of the software engineering process are evaluated the test specification process and the resulting tests should be assessed.
While consistency is defined as the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a component or system [IEG90], *consistency of a test* relates entirely to the test system. An example of the consistency check is evaluating whether the test pattern applied in the test is not empty.
Correctness is the degree to which a system or component is free from faults in its specification, design, and implementation [IEG90]. *Correctness of a test* is denoted in this thesis by the degree to which a test is free from faults in its specification, design and applied algorithms, as well as in returning the test verdicts. This definition is extended in comparison to *test correctness* provided by [ZVS$^+$07]. There it is understood as the correctness of the test specification w.r.t. the system specification or the test purposes, i.e., a test specification is only correct when it always returns correct test verdicts and when it has reachable end states.
Correctness of a test can be exemplified when the robustness of the test is considered, e.g., in MiLEST a check is made to see whether the tolerance limits applied in the assertions are high enough to let the test pass.

In this thesis, *consistency and correctness of a test* are mainly defined w.r.t. to the test scenarios specified applying the MiLEST method. Both of them can be assessed by application of the corresponding test quality (TQ) metrics.

Progress on the TQ metrics has been achieved by [VS06, VSD07], where static and dynamic metrics are distinguished. The static metrics reveal the problems of the test specification before

[41] Metrics related to structural model coverage are already well established in the common practice. They analyze path coverage, branch coverage, state coverage, condition coverage, cyclomatic complexity, MC/DC, etc. of a system model [ISTQB06, IEG90, MathSL].

its execution, whereas the dynamic metrics relate to the situation when the test specification is analyzed during its execution.
An example of a static consistency check is evaluating if at least one test for each requirement appears in the test specification, whereas a dynamic check determinates whether a predefined number of test cases has been really executed for every requirement.

Additionally, [ZVS$^+$07] define a TQ model as an adaptation of ISO/IEC 9126 [ISO04] to the testing domain. Its characteristics are taken into account in the upcoming analysis too.
These are:

- *test reusability* and *maintainability* that have already been discussed in the introduction to test patterns in Section 4.3;
- *test effectivity* that describes the capability of the specified tests to fulfill a given test purpose, including test coverage, test correctness, and fault-revealing capability, among others;
- *reliability* that reveals the capability of a test specification to maintain a specific level of work and completion under different conditions with the characteristic of maturity, thus consistency;
- *usability* that describes the ease to actually instantiate or execute a test specification, distinguishing understandability, and learnability, among others;
- finally, *efficiency,* and *portability*. These, however, are left out of the scope in this thesis.

7.2.2 Test Quality Metrics

For the purpose of this thesis several *TQ metrics* have been defined, based mainly on the functional relevance. In the following paragraphs, they are discussed and ordered according to the test specification phases supported by the MiLEST methodology. Obviously, the list is not comprehensive and can be extended in many directions.
Additionally, a classification of the presented TQ metrics in terms of the criteria defined in Section 7.2.1 is provided in Table 7.1.

Test data related quality metrics:

- *Signal range consistency* is used to measure the consistency of a signal range (specified in the *Signal Range* block) with the constraints put on this range within the preconditions or assertions at the VF level. It applies for SUT inputs and outputs. The consistency for inputs is implicitly checked when the variants of the test signals are generated. In other words, the test data generator informs about the inconsistencies in the signal ranges. The metric is used for positive testing.

- *Constraint correctness* – assuming that the signal range is specified correctly for every SUT interface, it is used to measure the correctness of constraints put on those signals within the preconditions or assertions at the VF level. If the ranges are violated, then the corresponding preconditions or assertions are not correct.

- *Variants coverage for a SigF* is used to measure the equivalence partitions coverage of a single SigF occurring in a test design. It is assessed based on the signal boundaries, partition points, and SigF type and uses similar methods to those for the generation of test stimuli variants. The metric is used for positive testing. The maximum number of variants for a selected SigF is equal to the sum of all possible meaningful variants. The metric can be calculated before the test execution by:

$$\text{Variants coverage for a SigF} = \frac{\#\,\text{of variants for a selected SigF applied in a test design}}{\#\,\text{of all possible variants for a selected SigF}} \tag{7.2}$$

The sign # means *'number of'*.

– *Variants coverage during test execution* is used to measure whether all the variants specified in the test design are really applied during the test execution. It returns the percentage of variants that have been exercised by a test. Additionally, it checks the *correctness* of the sequencing algorithm applied by the test system.

$$\text{Variants coverage during test execution} = \frac{\#\,\text{of variants applied during test execution}}{\#\,\text{of variants specified in a test design}} \tag{7.3}$$

– *Variants related preconditions coverage* checks whether the preconditions have been active as many times as the different combinations of test signal variants stimulated the test. It is calculated during the test execution.

$$\begin{aligned}&\text{Variants related preconditions coverage} = \\ &= \frac{\#\,\text{of variant combinations present in a given test data set being applied during test execution}}{\#\,\text{of activations of a selected preconditions set}}\end{aligned} \tag{7.4}$$

– *Variants related assertions coverage* is used to measure whether all the combinations of test signal variants specified within a given test data set (thus, generated from a corresponding preconditions set) and applied during a test cause the activation of the expected assertions set.

$$\begin{aligned}&\text{Variants related assertions coverage} = \\ &= \frac{\#\,\text{of variant combinations present in a given test data set being applied during test execution}}{\#\,\text{of activations of a selected assertions set}}\end{aligned} \tag{7.5}$$

– *SUT output variants coverage* is used to measure the range coverage of signals at the SUT output after the test execution. It is assessed based on the signal boundaries and partition points using similar methods like for the generation of test stimuli variants. The metric can be calculated by:

$$\text{SUT output range coverage} = \frac{\#\,\text{of the resulting variants for a selected output recognized after test execution}}{\#\,\text{of all possible variants for a selected output}} \tag{7.6}$$

- *Minimal combination coverage* is used to measure the coverage of combining the test stimuli variants according to the minimum criterion. 100% coverage requires that every variant of a SigF is applied at least once in a test step[42] (i.e., technically, it appears in the test data set). The metric is also called *each-used* or *1-wise* coverage.

$$\text{Minimal combination coverage} = \frac{\text{sum of the applied variant combinations satisfying 1-wise criterion}}{\text{sum of all possible combinations satisfying 1-wise criterion}} \tag{7.7}$$

Test specification related quality metrics:

- *Test requirements coverage* compares the number of test requirements covered by test cases specified in MiLEST test to the number of test requirements contained in a corresponding requirements document. It is calculated by:

$$\text{Test requirements coverage} = \frac{\text{\# of test requirements covered in a test design}}{\text{overall \# of test requirements}} \tag{7.8}$$

- *VFs activation coverage* is used to measure the coverage of the VFs activation during the test execution. This metric is related to the test requirements coverage, but one level lower in the MiLEST hierarchy. It is calculated as follows:

$$\text{VFs activation coverage} = \frac{\text{\# of VFs activated during test execution}}{\text{\# of all VFs present in a test design}} \tag{7.9}$$

- *VF specification quality* is used to assess the quality of an IF-THEN rule specification. In particular, it evaluates whether the test data generation algorithm is able to provide a reasonable set of test signals from the specified preconditions. In other words, a check is made to see whether the signals within preconditions of VFs are transformable into the test stimuli. The metric, though simplified, results from the discussion given initially by [MP07] (pg. 92) and continued in Section 5.2 on Modus Tollens [Cop79, CC00] rules application. If only the SUT outputs are constrained in the preconditions part of a VF, then no test stimuli can be produced from it. In that case, it is reasonable to modify (e.g., reverse) the VF contents. If the result equals 0 then the IF-THEN rule is not correct and must be reconstructed.

$$\text{VFs specification quality} = \frac{\text{\# of input signals applied in a preconditions set}}{\text{\# of all signals present in the preconditions set}} \tag{7.10}$$

- *Preconditions coverage* measures whether all the preconditions specified in the test design at the VF level have really been active during the test execution.

[42] Test step is a unique, non-separable part of a test case according to the nomenclature given in Section 5.4.2. The *test case* can be defined as a sequence of *test steps* dedicated for testing one single requirement.

$$\text{Preconditions coverage} = \frac{\text{\# of preconditions active during test execution}}{\text{\# of all preconditions specified in a test design}} \tag{7.11}$$

- *Effective assertions coverage* uses cause-effect analysis to determine the degree to which each effect is tested at least once. In other words, it reveals the number of implemented assertions being active during the test execution in relation to the number of all possible IF-THEN textual rules.

$$\text{Effective assertions coverage} = \frac{\text{\# of effects tested by an assertion}}{\text{overall \# of assertions specified in a test design}} \tag{7.12}$$

Test control related quality metric:

- *Test cases coverage* is used to measure the coverage of real activations of test cases. The sequence of test cases to be activated is specified in the test control unit. The metric is calculated by the formula:

$$\text{Test cases coverage} = \frac{\text{\# of the activated test cases}}{\text{\# of all test cases present in a test control design}} \tag{7.13}$$

Other TQ metrics: Additionally, dedicated metrics for different indirect testing activities can be defined. An interesting example results from the discussion about services on the interaction models in Section 5.6.

- *Service activation coverage* measures the number of services being executed in a test in relation to the number of all services that have been designed in the test specification. It is calculated by the formula:

$$\text{Service activation coverage} = \frac{\text{\# of the executed and evaluated services (or parts of services)}}{\text{\# of all specified services (or parts of services) in hySCts}} \tag{7.14}$$

Realization: A few of the mentioned TQ metrics have been realized in the prototype. These are implemented either as SL subsystems or as ML functions. For instance, implementation of the VFs activation coverage is based on computing the number of local verdicts for which the value has been different from *none* or *error* in relation to the number of all VFs. The situation is illustrated in Figure 7.4.

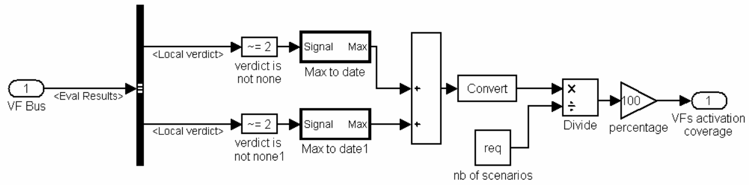


Figure 7.4: Implementation of the VFs Activation Coverage Exemplified for Two of Them.

Another example is a ML function called *input_coverage()* that lists the results of the metric called *variants coverage for a SigF* for every SUT input signal.

Test modeling guidelines check: Furthermore, apart from the measurements realized with the help of TQ metrics, consistency of the test specification may be checked statically by application of the test modeling guidelines. The modeling guidelines may be either company specific or defined by some institutional bodies, e.g., the Motor Industry Software Reliability Association (MISRA™) [Mis]. The technical realizations of modeling guideline checkers for system design are presented in [NSK03, AKR+06, ALS+07, FG07] using graph transformations or OCL.

Some ad-hoc test modeling guidelines rules for the test specification developed in MiLEST are given below:

- Assertions block in a VF cannot be empty
- Preconditions block in a VF cannot be empty
- Test data generation unit cannot be empty
- Hierarchical structure for TDGen and TSpec units should be preserved
- Number of SigFs found in the preconditions of a VF should be equal or higher than the number of inputs in the preconditions
- Number of SigFs found in the assertions of a VF should be the same as the number of the PAS blocks
- One PS block should be present in the preconditions of a VF
- PAS inputs are signals that pass on from the comparison blocks
- Comparison blocks must have the reference data
- *'TestInfo'* bus is built by the set of signals: {Local verdict, Expected lower limit, Expected upper limit, Actual data, Delay}
- The names of some objects present in the test model, e.g., *'TestInfo'* should remain fixed.

7.2.3 Classification of the Test Quality Metrics

Similar to separate test activities needing to be performed for both functional requirements approval (i.e., black-box test) and structural model validation (i.e., white-box test), the same applies to quality considerations. Hence, already [Con04a] claims that *system model coverage* should be used in combination with the other functionality-related metrics.

A possible realization that measures structural coverage at the model level is provided by the Model Coverage Tool in Simulink Verification and Validation [SLVV]. There, model coverage is a metric collected during simulation delivering information about model objects that have been executed in simulation. In addition, it highlights those objects that have not yet been tested. The metric assesses the completeness of the test. A generic formula used for its calculation is given by [ZVS+07]:

$$\text{System model coverage} = \frac{\text{achieved coverage of system model structure}}{\text{possible coverage of system model structure}} \tag{7.15}$$

Furthermore, the discussion on quality is given by [Leh03, Con04a] w.r.t. their concrete test approaches. Thus, for Time Partitioning Testing – *cost/effort needed for constructing a test data set*, *relative number of found errors in relation to the number of test cases needed to find them* – are named as examples. [Con04a] adds *coverage of signal variants combinations – CTC_{max}, CTC_{min}* initially introduced in [Gri95] for Classification Tree Method – CTM (cf. *minimal combination coverage* in Table 7.1). Obviously, both authors distinguish *requirements coverage* (cf. *test requirements coverage* in Table 7.1) as an important measure too.

In Table 7.1 all the TQ metrics that have been defined in Section 7.2.2 and in this section are classified according to the criteria provided in Section 7.2.1.

Table 7.1: Classification of MiLEST Test Quality Metrics.

Classification Criteria / *TQ Metrics*	*TQ Model*	*Assessability Phase*	*Consistency of test, correctness of test*
Test data related:			
Signal range consistency	- Reliability (maturity)	- Static	- Consistency of test
Constraint correctness	- Reliability (maturity) - Effectivity (test correctness)	- Static	- Correctness of test
Variants coverage for a SigF	- Reliability (maturity) - Effectivity (test coverage)	- Static	- Consistency of test
Variants coverage during test execution	- Effectivity (test coverage) - Effectivity (test correctness)	- Dynamic	- Correctness of test
Variants related preconditions coverage	- Effectivity (test coverage) - Effectivity (test correctness)	- Dynamic	- Correctness of test
Variants related assertions coverage	- Effectivity (test coverage) - Effectivity (test correctness)	- Dynamic	- Correctness of test
SUT output variants coverage	- Effectivity (test coverage)	- Dynamic	- Does not apply
Minimal combination coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test
Test specification related:			
Test requirements coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test
VFs activation coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test

VF specification quality	- Effectivity (test correctness) - Reliability (maturity) - Usability (understandability) - Usability (learnability)	- Static	- Correctness of test - Consistency of test
Preconditions coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test
Effective assertions coverage	- Effectivity (test coverage) - Effectivity (fault revealing ability)	- Dynamic	- Consistency of test
Test control related:			
Test cases coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test
Others:			
Service activation coverage	- Effectivity (test coverage)	- Dynamic	- Consistency of test
System model coverage	- Effectivity (test coverage)	- Dynamic	- Does not apply
Cost/effort needed for constructing a test data set	- Does not apply	- Static	- Does not apply
Relative number of found errors in relation to the number of test cases needed to find them	- Effectivity (fault revealing ability)	- Dynamic	- Does not apply
Coverage of signal variants combinations – CTC_{max}, CTC_{min}	- Effectivity (test coverage)	- Dynamic	- Consistency of test

7.3 The Test Quality Metrics for the Case Studies

The test quality metrics can be calculated during the test specification phase when applied on the design or during and after the test execution. In this section, the practical relevance of the metrics is illustrated for the selected case studies that have been provided in Chapter 6. Nevertheless, in the following only a few post-execution metrics are computed since the test-design-based ones are covered almost 100%. This applies, in particular, to the test-specification-related metrics since the necessary condition to conclude on further steps in the test process is a complete and exhaustive definition of VFs in MiLEST models. This is not the case for the test data generation, though, as here the TDG implementation details matter as well.

For illustration purposes v*ariants coverage for SigF* and *SUT output variants coverage* are calculated for the MiLEST models of all the case studies in the form they were in when they were introduced in Chapter 6. Thereby, it is shown what progress can be achieved and concluded, and how much effort is needed for different types and test levels so as to obtain an expected level of advance.

The metrics investigating the cost or power of the method in terms of finding the errors will not be analyzed in this thesis since the author can neither objectively assess the efforts spent on using the proposed test method, nor possess enough statistics regarding more examples of MiLEST application.

7.3.1 Pedal Interpretation

In the pedal interpretation case study all the metrics related to the test data, test specification, and test control have reached their maximal values (cf. Table 7.2); therefore, they will not be discussed in detail here. Further on, the metric called *service activation coverage* is not calculable for this example since only the component level test is regarded here.

Table 7.2: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the Pedal Interpretation.

Direction	*Name*	*Domain*	*Partition Point*	*Achieved Coverage*
Input Signal	v_act	[-10, 70]	{0}	100 %
	phi_Acc	[0, 100]	{5}	100 %
	phi_Brake	[0, 100]	{5}	100 %
Output Signal	Acc Pedal	[0,1]	–	100 %
	Brake Pedal	[0,1]	–	100 %
	T_des Drive	[-8000, 2300]	{0}	100 %
	T_des Brake	[0, 4000]	{0}	100 %

7.3.2 Speed Controller

In contrast to the previous model, the test of speed controller covers fewer issues. For instance, v*ariants coverage for SigF* and *SUT output variants coverage* reveal lower levels as given in Table 7.3. The reason is that the test data variants have not been generated yet. Instead, the case study has provided the way for how to automate the mechanism, which is sequencing the test steps or test cases. The values of the metrics indicate that further efforts are needed so as to complete the test design in terms of data selection.

Table 7.3: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the Speed Controller.

Direction	*Name*	*Domain*	*Partition Point*	*Achieved Coverage*
Input Signal	v_des	[11, 70]	–	33 %
	CCMode	[]	{0,1,2,3}	25 %
	Selection	[]	{1,2,3}	33 %
Output Signal	v_act	[-10, 70]	{0}	40 %

7.3.3 Adaptive Cruise Control

Similar considerations to those for the previous case study apply to adaptive cruise control (ACC). The values obtained for v*ariants coverage for SigF* and *SUT output variants coverage* (see Table 7.4) indicate that there is still plenty to do in terms of test data generation. Here, the reason for such results is that only one path of the *High level hybrid Sequence Chart for testing (HhySCt)* has been traversed during the test data generation (cf. Section 5.6).

Table 7.4: Variants Coverage for SigF and SUT Output Variants Coverage Exemplified for the ACC.

Direction	*Name*	*Domain*	*Partition Point*	*Achieved Coverage*
Input Signal	v_aim	[-10, 70]	{0}	20 %
	phi_Acc	[0, 100]	{5}	100 %
	phi_Brake	[0, 100]	{5}	80 %
	LeverPos	[]	{0,1,2,34}	11 %
	DistanceFactor	[0,1]	–	33 %
	v_des	[11,70]	–	33 %
Output Signal	v_act	[-10,70]	{0}	67 %
	Torque_brake	[0,4000]	–	100 %
	Torque_engine	[-8000, 2300]	{0}	60 %

The metrics applicable for ACC relate additionally to further aspects. For example, the *service activation coverage* achieves coverage of 14% since only one path of the corresponding *HhySCt* diagram is included in the test execution.

7.3.4 Concluding Remarks

An important remark is that the values of metrics calculated throughout the case studies have been differentiated on purpose. In other words, the test design of the speed controller and ACC are presented as incomplete so as to explain the extracted ideas behind different elements of MiLEST more clearly, show the weight of the introduced concepts in comparison and reveal the need for application of the test quality metrics.

In reality, the test designs are much more complete. However, it would make no sense to prove that all of them yield maximal values of metrics.

Furthermore, the only quantitative results that the author of this thesis is able to relate to are included in the work of [Con04a], where similar case studies have been applied. The obtained values of the metrics in the context of the test data selection reveal that the automatically generated test cases produce very similar test coverages w.r.t. different aspects as a manual specification using classification trees executed in MTest.

The component level test in MiLEST may be fully automated, whereas the integration level test needs manual refinement. Nevertheless, it is still better than simply a manual test stimuli selection.

Concerning the test evaluation considerable progress has been achieved in this thesis since it works automatically regardless of which input is being applied. In other words, if the test evaluation has been specified once, it runs independently of the SUT stimulation, in contrast to the MTest approach.

7.4 Quality of the Test Strategy

In the upcoming paragraphs the criteria defining an efficient and effective test strategy given by [Leh03, Con04a][43] are analyzed and modified in such a way that the test dimensions (defined in Section 2.3) and test categories (provided in Section 3.1) constituting the aims of this thesis, remain the primary focus.

The criteria defined by [Leh03] are:

1. automation of the test execution
2. consistency throughout different execution platforms
3. systematic test data variants selection
4. readability
5. reactive test support
6. real-time and continuous behavior support

Whereas [Con04a] indicates the following as the most important:

1. possibility to describe different signal categories (related to (6) of [Leh03])
2. test abstraction and type of description means (related to (4) of [Leh03])
3. expressiveness (related to (6) of [Leh03])
4. coverage criteria support (related to (3) of [Leh03])
5. embedding in the existent methodology (related to (2) of [Leh03])
6. tool support (related to (1) of [Leh03])
7. wide-spreading in the domain

Based on the above, the criteria for MiLEST aiming for high-quality test strategy and methodology are listed:

1. automation of the test specification process and test execution (cf. Section 4.4, Sections 5.2 – 5.4), similar to (1) of [Leh03]
2. systematic selection of test signals and their variants (cf. Section 5.3), similar to (3) of [Leh03]
3. readability, understandability, ease of use (cf. test patterns in Chapters 4 – 5), similar to (4) of [Leh03]
4. reactive test support (cf. Section 5.4), similar to (5) of [Leh03]
5. real-time, discrete and continuous behavior support (cf. Chapter 4), similar to (6) of [Leh03]
6. abstraction, test patterns support (cf. Chapters 4 – 5)
7. high quality, correctness, and consistency of the resulting *test model* (cf. this chapter), similarly as (4) of [Con04a]
8. support of signal-feature – oriented data generation and their evaluation (cf. Section 4.1 – 4.2), related to (1,3) of [Con04a])

The embedding of the test approach within the model-based development paradigm (as (5) of [Con04a]) and tool support (as (6) of [Con04a]) are obvious criteria that any new technical proposal must fulfill.

[43] The two approaches have been selected since they have already been in use at Daimler AG for the last few years [KHJ07].

An undeniable fact is that the automotive industry aims at transferring innovative model-based testing technologies from research into industrial practice. However, eventually any company is interested in reducing cost for testing by reaching higher test coverage simultaneously, which subsequently enables more errors to be detected whatever test techniques are used [DM_D07, D-Mint08]. Therefore, a very crucial industry-driven research question is the quality of the test cases and test design. This can be measured by means of different quality metrics as discussed in Section 7.2.

7.5 Limitations and Scope of the Proposed Test Method

The main limitations of the MiLEST method and its realization result from the context of the applied test strategy and are outlined below.

- Support of different test execution platforms for the proposed method has not been sufficiently explored yet. Consequently, the consistency throughout the platforms cannot be considered. System engineers use the models developed on MiL level for HiL, SiL, and PiL platforms, though. Interesting research questions arise asking to what extent the concrete test cases should be reused on different levels.
- Real-time properties on the run-time execution level in the connection with hardware devices in the sense of scheduler, RTOS, priorities, and threads have not been investigated. Also, the test system itself has not been designed to be real time (cf. Section 3.3).
- Distribution of the method within the automotive domain is not yet possible since MiLEST is not a ready-to-use product, but rather an extendable prototypical realization. If customized further, it might be applied on a large scale in the future. These and other issues are currently under discussion with the industrial partners [D-Mint08]. Moreover, the tendency to handle continuous signals based on their features and predicates is a promising one as discussed by [ZSM06, MP07, GW07, SG07, Xio08] (cf. Section 4.5).
- In addition, the complexity of the method and the learning curve influenced by the learning ability of the test engineer cannot be assessed in a straightforward manner. Nevertheless, the analysis of a simple questionnaire that has been filled out by executive managers of software- and test-related projects and is attached in Appendix D, gives an overview of the acceptance rate for different test modeling techniques. It reveals that the CTM and sequence-diagrams-based testing are still the most widespread test methods in the industry.

While soundness of the proposed methodology is analyzed by application of the test metrics, its completeness deserves some considerations. It is applicable to causal systems of either continuous or discrete nature, or the combination of both of them, alternatively described by time constraints. The methodology and test development process could be reused also for other types of SUTs. However, the implementation restricts MiLEST to these specified in the ML/SL/SF environment.

From the perspective of the technical domain, MiLEST suits any area where hybrid embedded software is developed. Hence, besides the automotive one, it can be deployed in avionics, aerospace, rail, or earth information systems, after some adjustments.

Further discussion on the restrictions and challenges of MiLEST methodology will be given in Section 8.1. Despite the listed limitations, it is believed that the novel test paradigm described in Chapter 4 and the advantages resulting from the implications of the applied strategy discussed in Chapters 3, 5, 6 represent enough decisive factors to pay attention to the MiLEST.

7.6 Summary

The quality of the test method proposed throughout this thesis has been the main subject of this chapter. Prototype must have been supported since it is a reliable proof of concept for validation of any newly developed approach. Further on, the test models designed in MiLEST have been explored so as to define a number of *test quality metrics*. By that, the fourth research question introduced in the first chapter of this thesis has been adressed. *The quality* of the resulting tests and the test method itself can be assessed by application of those *metrics*.

In Section 7.1, the prototypical realization has been discussed. The ML scripts, callback functions, transformation functions, and the MiLEST library have been attached to this thesis and listed in Appendix E. Section 7.2 has served as a backbone for this chapter since here, the quality metrics of the test specification process, test design, and the resulting test cases have been investigated. They have been classified according to the predefined criteria and contributed as input for Section 7.3. There, they have been calculated for the case studies that had been introduced in Chapter 6. Afterwards, in Section 7.4, the commonly known test strategies have been reviewed and compared with the strategy proposed in this thesis. Based on the results, the challenges, scope and limitations of MiLEST have been indicated in Section 7.5.
This discussion will be continued and summarized in the next chapter in order to illustrate the remaining research potential and its new directions in relation to the achievements gained throughout this work.

8 Summary and Outlook

"Perfection is achieved, not when there is nothing more to add,
but when there is nothing left to take away."

- Antoine de Saint-Exupéry

This is the last chapter of this thesis. It is divided into three sections, the first of which contains a general summary, including a discussion on the *problems* and *challenges* that have been introduced in Section 1.2. Then, the outlook part emphasizes two aspects – future work, advantages and limitations of the test method proposed herewith, as well as general trends of the quality assurance (QA) for embedded systems (ES). Afterwards, in the last section, indirect influences of the contributions presented here are outlined.

8.1 Summary

The first part of this thesis has dealt with general information on its topics. Hence, Chapter 1 has introduced the motivation, scope and contributions of this work. Also, the structure has been provided there and a roadmap has been discussed.

Then, in Chapter 2, the backgrounds on ESs and their development have been described. Herewith, the fundamentals on the control theory and electronic control units have been provided. Besides, model-based development concepts applied in the automotive domain have been introduced. Also, basic knowledge on the MATLAB/Simulink/Stateflow (ML/SL/SF) framework and testing concerns has been given. The testing has been categorized by the dimensions of test scopes, test goals, test abstraction, test execution platform, test configuration, and test reactiveness for the needs of this thesis. Functional, abstract, executable, and reactive MiL level tests have been put in the center of attention.

In Chapter 3, analysis of the related work on model-based testing (MBT) has been provided. MBT *taxonomy* has been elaborated, extended, and presented on a diagram, where the test generation, test execution, and test evaluation were in focus. Then, the current testing situation in the automotive domain has been reported. A comprehensive classification of the MBT solutions has been attached in Appendix A. Finally, based on the analysis of challenges and limitations of the existing approaches (cf. *first challenge* given in Section 1.2), characteristics of the method proposed in this thesis have been briefly elaborated on.

Next, the intention of the second part of this thesis has been to introduce the Model-in-the-Loop foe Embedded System Test (MiLEST) method. In Chapter 4, a new way of signal description by application of a signal feature has been investigated. The features have been classified, and their generation and detection have been realized (cf. *second challenge*). Furthermore, a test specification process, its development phases and artifacts have been discussed. Also, test patterns have been described and attached to this thesis in Appendix B.

Chapter 5 has been based on the previous chapter. Here, the methodological and technical details of MiLEST (cf. *second challenge*) have been explained. MiLEST extends and augments ML/SL/SF for designing and executing the test. It bridges the gap between system and test engineers by providing a means to use SL/SF language for both SUT and test specification (cf. *first challenge*).
Then, the classification of signal features has been recalled so as to describe the architecture of the test system. Thus, different abstraction levels of the test system have been provided (cf. *second challenge*). They were denominated relating to the main activities performed at each level. The test harness level included the patterns for test specification, test data generation, and test control. Then, the test requirements level appeared. It has been followed by the test case and validation function levels. Afterwards, the feature generation and feature detection have been elaborated. In Appendix C a consolidation of the hierarchical architecture has been attached.
Furthermore, in Chapter 5, different options for the test specification have been reviewed. The importance of the test evaluation has been emphasized. The automatic generation of the test data has been presented. By means of concrete generic transformation rules, the derivation of test signals from the validation functions (VFs) has been formalized. Similarly, the generation of signal variants has been investigated. Combination strategies for test case construction have been outlined and sequencing of the generated variants at different levels has been reported. The concept of reactive testing and the test control have been summarized too. All these contributed to the definition of a test development process and automation of some of its phases (cf. *third challenge*).

Figure 8.1 summarizes the conceptual contents of Chapters 4 – 5, positioning MiLEST and its main value disposers.

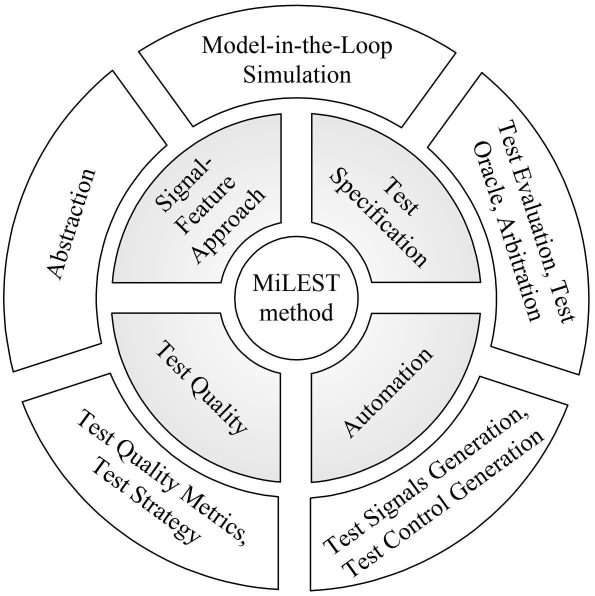

Model-in-the-Loop Simulation
Test Evaluation, Test Oracle, Arbitration
Test Signals Generation, Test Control Generation
Test Quality Metrics, Test Strategy
Abstraction
Signal-Feature Approach
Test Specification
Test Quality
Automation
MiLEST method

Figure 8.1: Overview of the Thesis Contents.

The advantages of MiLEST combination with the *hybrid Sequence Charts for testing (hySCts)* at the integration level have been discussed. The details of the approach from the methodological and technological perspective have been given.

Due to the application of a graphical representation of the requirements in the form of *hySCts*, the contents of the resulting QA process can be understood more easily by different stakeholders. This fact has been confirmed by the analysis of a simple questionnaire that has been filled out by several executive managers of software- and test-related projects (Appendix D). Herewith, the sequence diagrams have been selected as the most widespread methods in the industry.

Then, the high level *hySCts* have been reused to generate the concrete test data systematically and semi-automatically at the model integration level test (cf. *third challenge*).

In the third part of this thesis the practical relevance of the presented concepts has been proven. In Chapter 6, three case studies related to the functionality of an adaptive cruise control (ACC) have been analyzed. The entire system under test has been initially realized at Daimler AG, which has provided the functional requirements and the model. The extracted pedal interpretation example, used for the illustration of component level test, has showed the application of the test specification and data generation algorithms in detail. Then, the speed controller has provided the basis for component in the loop level test, proving mainly the feasibility of reactive testing concepts. Finally, ACC itself has been investigated. Here, the *hySCt* models have been designed. Then, the model integration level test has been designed. The test data have been retrieved by assuring the coverage of the selected test objectives.

Chapter 7 explored the test models designed in MiLEST so as to define a number of test quality metrics that, in turn, enabled to evaluate the resulting tests (cf. *fourth challenge*). The values of the metrics for the test specification process, test design, and the resulting test cases have been

calculated based on the case studies from the previous chapter. In addition, the prototypical realization has been discussed and summarized in Appendix E. Afterwards, the commonly known test strategies have been reviewed and compared with the strategy proposed in this thesis. Also, the challenges and limitations of MiLEST have been indicated.
This chapter serves as a close to this thesis and as such is self-explanatory. Thus, summing up, MiLEST constitutes a comprehensive QA framework (cf. Figure 8.1) enabling a full test specification for ES based on the ready-to-use test patterns.

The MiLEST *signal-feature* approach provides the essential benefit of describing signal flows, their relation to other signal flows, and/or to discrete events on an abstract, logical level. This prevents not only the user from error on too-technical specifics, but also enables test specification in early development stages. The absence of concrete reference signals is compensated by a logical description of the signal flow characteristics.

In addition, MiLEST automates the systematic test data selection and generation, providing a better test coverage (e.g., in terms of requirements coverage or test purpose coverage) than manually created test cases.

Furthermore, the automated test evaluation reveals considerable progress, in contrast to the low level of abstraction and the manual assessment means used in existing approaches. The tester can immediately retrieve the test results and is assured about the correct interpretation of the SUT reactions along the tests.
The signal evaluation used in the automated test evaluation can even run independently of the SUT stimulation. The signal evaluation is not based on the direct comparison of SUT signals and previously generated concrete reference signals. Instead, it offers an abstract way for requirements on signal characteristics. The signal evaluation is particularly robust and can be used in contexts other than testing, e.g., for online monitoring.

Furthermore, the test specification enables the SUT requirements to be traced. The manner how it is defined gives the possibility to trace root faults by associating local test verdicts to them, which is a central element in fault management of embedded systems.

Finally, the MiLEST test quality metrics reveal the strengths of the approach by providing high test coverage in different dimensions and good analysis capabilities. The MiLEST projects demonstrated a quality gain of at least 20%.

8.2 Outlook

Due to the application of MiLEST the test engineer needs considerably less effort in the context of test generation and test execution, concentrating on the test logic and the corresponding test coverage. By that, the cost of the entire process of software development is definitely cut. However, there is still plenty of work concerning MiLEST future achievements.

Starting with the conditional rules utilized within the test specification and the discussion provided in Section 5.2 on the problem of ordering the SUT inputs and outputs constraints, an automatic transformation of the incorrect functions (e.g., *IF constrained output THEN constrained input*) into the reverse, based on the transposition rule [CC00] could be easily realized.

Then, the issue of handling the SUT outputs existing in the preconditions of a VF (e.g., *IF constrained input AND constrained output THEN constrained output*) has been only partially solved. A check is made to see whether the test data generator has been able to produce the meaningful signals from such a specification. If this is not the case, a manual refinement is needed in the Initialization/Stabilization block at the test case level. An automatic way of relating the functional test cases and test sequences to each other can be a struggle to find, though.

The test stimuli generation algorithms can still be refined as not all the signal features are included in the realization of the engine. Also, they could be enriched with different extensions applying, in particular, a constraint solver and implicit partitions as discussed in Section 5.4.1. There is further work regarding the negative testing as well. Here, the test data generation algorithm can be extended so as to produce the invalid input values or exceptions. However, the test engineer's responsibility would be to define what kind of system behavior is expected in such a case.
An interesting possibility pointed out by [MP07] would be to take advantage of the reactiveness path for optimizing the generated test data iteratively. In that case, the algorithm could search for the SUT inputs leading to a fail automatically (cf. evolutionary algorithms [WW06]).

Moreover, since software testing alone cannot prove that a system does not contain any defects or that it does have a certain property (i.e., that the signal fulfills a certain signal feature), the proposed VFs could be a basis for developing a verification mechanism based on formal methods in a strict functional context [LY94, ABH$^+$97, BBS04, DCB04]. Thus, the perspective of mathematically proving the correctness of a system design remains open for further research within the proposed QA framework.

Besides the test quality metrics analyzed in Chapter 7, other criteria may also be used to assess the proposed test method, the following being only some of them:

- the efficiency of faults detection – aimed to be as high as many VFs are designed under the assumption that the corresponding test data are generated and applied properly
- the percentage of test cases / test design / test elements reusability
- time and effort required for the test engineer to specify the test design, which is relatively low only for persons knowing the technologies behind the concepts
- the percentage of the effective improvement of the test stimuli variants generation in contrast to the manual construction.

Regarding the realization of MiLEST prototype, several GUIs (e.g., for transformation functions, for quality metrics application, for variants generation options, or for the test execution) would definitely help the user to apply the method faster and more intuitively.

Furthermore, the support of different test execution platforms for the proposed method has not been sufficiently explored yet. Interesting research questions concern the extent to which the concrete test cases could be reused on various execution levels. Consequently, real-time properties on the run-time execution level [NLL03] in the sense of scheduler, RTOS, priorities, or threads have not been considered.

Then, the Testing and Test Control Notation (TTCN-3) is worth mentioning since the advantages of this standard technology have already been recognized by the automotive industry.

AUTOSAR and MOST [ZS07] decided to use TTCN-3 for the definition of functional tests. Currently, TTCN-3 for ESs is under development [Tem08, GSW08].
Investigation in the context of transformations from UML Testing Profile for Embedded Systems [Liu08] into executable MiLEST test models and further on, into TTCN-3 for ESs would be an interesting approach enabling the test specification to be exchanged without losing the detailed information since all three approaches are based on a similar concept of signal feature. This would give the advantage of having a comprehensive test strategy applicable for all the development platforms.

Moreover, an interesting option would be to investigate the potential of MiLEST for testing the AUTOSAR elements having in mind that an analysis of Simulink itself is already included in the AUTOSAR standard.

8.3 Closing Words

There is still plenty of research potential resulting from this thesis, in general. The authors of [Hel$^+$05] consider the multidisciplinary character of the embedded domain as a challenge that should not be neglected. Indeed, it is necessary in software development for ES to consider several aspects together. Hardware/software field alike, technological progress and economic success profit highly from human abilities of cooperation and the social environment.
The argumentation applies to the QA disciplines too. There should be a link or at least a conceptual common understanding forwarded from MiLEST to other test types, e.g., structural test, performance test, or robustness test so as to position all the QA activities in the development process.
Generally, the concept of working together with any other parties holds at every level of interactions, starting from software development, through assuring the quality and safety of the resulting product, to the real effects while running this product (e.g., fuel consumption, carbon dioxide (CO_2) emissions[44]). Europe has already shown interest towards environmentally friendly hybrid cars. The market for hybrid ECUs used for engine management is expected to witness a steady growth across the forecast period with units and revenues expected to reach 0.7 million and € 26.1 million by 2015, respectively [FS08]. Hence, similarly as the trends of sustainable development[45] indicate, this thesis has aimed at accomplishing local actions that think globally in parallel.

Moreover, the effects of any progress achieved in the research on fuel cells and hybrid cars should be included in analyzing the role of software and new paradigms of its QA to shape our common future.
ESs ease environmental research and enable many new investigations on the measurement and tracking of diverse data, like weather and climate data. Sensors are used to identify early indicators of earthquakes, volcanic eruptions, or floods. Wireless sensor networks are used to observe

[44] According to the recent trends in new car purchases in European Union, the average car sold over the period 1995-2004 experienced a surge in power of 28% while CO_2 emissions decreased by 12% [EU06].

[45] Sustainable development is the development that meets the needs of the present without compromising the ability of future generations to meet their own needs [Bru87].

animals and to track changes of ecosystems in order to better understand these changes and examine the effects of nature conservation activities [Hel^{+}05]. Hence, the QA techniques for such constellations are important issues that will be dealt with in the near future.

The main wish of the author is that at least some parts of this thesis will be reused and the subject will be explored further on, e.g., based on one of the points from the outlook or in another domain. The author has already made some effort to enable such progress.

Glossary

Arbitration algorithm – The role of the *arbitration* mechanism is to extract the *overall verdict*, being a single common verdict for the entire test from the structured test results. Single requirement-related verdicts or local verdicts for every validation function can also be obtained. There is a default arbitration algorithm ordering the verdicts according to the rule: none < pass < fail < error.

Assertions set – An *assertions set* consists of at least one extractor for signal feature or temporally and logically related signal features, a comparator for every single extractor, and at least one unit for preconditions and assertions synchronization.

Hierarchical architecture of the test system – A leveled structure for designing the test system that includes the means for specification of test cases and their evaluation. It comprises several abstraction levels (test harness level, test requirement level, test case – validation function level, feature generation – feature detection level) that can be built systematically.

Model-based testing – *Model-based testing* is testing in which the *entire test specification* is derived in whole or in part from both *the system requirements and a model* that describe selected functional aspects of the SUT. In this context, the term *entire test specification* covers the *abstract test scenarios* substantiated with the concrete sets of test data and the expected SUT outputs. It is organized in a set of test cases.

Model-in-the-Loop for Embedded System Test (MiLEST) toolbox – A *toolbox* (i.e., library in the form of set of test patterns and functions) defined in MATLAB/Simulink/Stateflow environment that enables a hierarchically organized test system to be designed and the resulting test cases to be executed.

Preconditions set – A *preconditions set* consists of at least one extractor for signal feature or temporally and logically related signal features, a comparator for every single extractor, and one unit for preconditions synchronization.

Signal feature – A *signal feature* (also called signal property in the literature) is a formal description of certain defined attributes of a signal. It is an identifiable, descriptive property of a signal. It can be used to describe particular shapes of individual signals by providing means to address abstract characteristics of a signal. A *signal feature* can be predicated by other signal features, temporal expressions, or logical connectives.

Signal-feature evaluation – A *signal-feature evaluation* is an assessment of a signal based on its features. It consists of a preprocessing phase, extraction phase, where a *signal feature* of interest is detected to be compared with the reference value. Finally a verdict is set.
It is also called *signal evaluation*.

Signal-feature generation – A *signal-feature generation* is an algorithm, where feature is generated over a selected signal and the parameters are swept according to a predefined algorithm. The actual generation of every single *signal feature* must include its specific characteristics.
It is also called *signal generation*.

System model – A *model of an SUT* in the form of a block executable in MATLAB/Simulink/Stateflow environment, whereat the SUT represents a software-intensive embedded system.

Test case – A *test case* is a set of input values, execution preconditions, expected results, and execution postconditions, developed for a particular test objective so as to validate and verify compliance with a specific requirement. The *test case* can be defined as a sequence of *test steps* dedicated for testing one single requirement.

Test configuration – A *test configuration* is determined by the chosen SUT, additional components (e.g., car model), the initial parameters that must be set to let this SUT run and a concrete test harness.

Test control – A *test control* is a specification for the invocation of test cases assuming that a concrete set of test cases within a given test configuration exist.

Test design – A graphical and executable *design* modeled applying MiLEST notation. It comprises the entire test specification including the concrete test cases.
It is also called a *test model*.

Test dimensions – Tests can be classified in different levels, depending on the characteristics of the SUT and the test system. The *test dimensions* aimed at in this thesis are test goals, test scope, test abstraction, test reactiveness, and test execution platform.

Test evaluation – *Test evaluation* is a mechanism for an automatic analysis of the SUT outputs so as to decide about the test result. The actual SUT results are compared with the expected ones and a verdict is assigned. It is located in the test specification unit in MiLEST test model. It includes the arbitration mechanism and is performed online, during the test execution.

Test harness – The *test harness* pattern refers to the design level and is defined as an automatically created test frame including the generic hierarchical structure for the test specification. Together with the test execution engine (i.e., Simulink engine) it forms a *test harness*.

Test oracle – A *test oracle* is a source to determine the expected SUT results so as to decide about the test result. It is located in the test specification unit in the MiLEST test model.

Test pattern – *Test patterns* represent a form of reuse in the test development in which the essences of solutions and experiences gathered in testing are extracted and documented so as to enable their application in similar contexts that might arise in the future.
In this work the patterns for test harness, test specification, test data generation, signal-feature generators, signal-feature extractors, test evaluation, and test control are discussed.

Test process – The fundamental *test process* comprises planning, specification, execution, recording (i.e., documenting the results), checking for completion, and test closure activities (e.g., rating the final results).

Test quality – *Test quality* constitutes a measure for the test completeness and can be assessed on different levels, according to different criteria, applying metrics defined upon them.

Test quality metric – A *metric* is the measure applied on the test specification so as to estimate its test quality according to some predefined criteria.

Test specification – *Test specification* (TSpec) comprises abstract test scenarios describing the expected behavior of the SUT when a set of conditions are given. It consists of several test requirements which are boiled down to validation functions. It serves as an input for test data generation unit.
It is also called *test specification design* or *test specification model.*

Test step – A *test step* is derived from one set of preconditions from a validation function. It is related to the single scenario defined in the validation function within the test specification unit. Thereby, it is a unit-like, non-separable part of a test case.

Test suite – A *test suite* is a set of several test cases for a component or SUT, where the post-condition of one test is often used as the precondition for the next one. The specification of such dependencies takes place in the test control unit in the proposed test framework. It is particularly important in the context of integration level test, where the test cases depend on each other.

Validation function – A *validation function* defines the test scenarios and test evaluation based on the test oracle in a systematic manner. It serves to evaluate the execution status of a test case by assessing the SUT observations and/or additional characteristics/parameters of the SUT. It is created following the requirements by application of an IF – THEN conditional rule.

Verdict – A *verdict* is the result of an assessment of the SUT correctness. Predefined *verdict* values are pass, fail, none, and error. Verdict may be computed for a single validation function (i.e., assertions set), requirement, test case, or entire test.

Acronyms

A

A	–	Activation signal in PS
ABS	–	Antilock Braking System
ACC	–	Adaptive Cruise Control
ACM	–	Association for Computing Machinery
ADAS	–	Advanced Driver Assistance Systems
API	–	Application Programming Interface
ATG	–	Automatic Test Generator
AUTOSAR	–	Automotive Open System Architecture

C

CAN	–	Controller Area Network
CAGR	–	Compound Annual Growth Rate
CbyC	–	Correct by Construction
CLP	–	Constraint Logic Programming
CSREA	–	Computer Science, Research, Education, and Application
CTM	–	Classification Tree Method

D

DAG	–	Daimler AG
DC	–	Direct Current
DESS	–	Software Development Process for Real-Time Embedded Software Systems
D-MINT	–	Deployment of Model-based Technologies to Industrial Testing

E

ECU	–	Electronic Control Unit
ES	–	Embedded System(s)
ETSI	–	European Telecommunication Standardization Institute

F

FMEA	–	Failure Mode and Effects Analysis
FOKUS	–	Fraunhofer Institute for Open Communication Systems
FSM	–	Finite State Machine
FTA	–	Fault Tree Analysis

G

GDP	–	Gross Domestic Product
GUI	–	Graphical User Interface

H

HDL	–	Hardware Description Language
HhySCt	–	High level hybrid Sequence Chart for testing
HiL	–	Hardware-in-the-Loop
HW	–	Hardware
hySC	–	hybrid Sequence Chart
hySCt	–	hybrid Sequence Chart for testing

I

IEC	–	International Electrotechnical Commision
IEEE	–	Institute of Electrical and Electronics Engineers, Inc.
IFIP	–	International Federation for Information Processing
INCOSE	–	International Council on Systems Engineering
ISBN	–	International Standard Book Number
ISO	–	International Organization for Standardization
ISTQB	–	International Software Testing Qualifications Board
ITEA	–	Information Technology for European Advancement

ITU-T	–	International Telecommunication Union – Telecommunication Standardization Sector

J

JUMBL	–	Java Usage Model Builder Library

L

LIN	–	Local Interconnect Network

M

MAAB	–	The MathWorks Automotive Advisory Board
MaTeLo	–	Markov Test Logic
MATT	–	MATLAB Automated Testing Tool
MBD	–	Model-based Development
MBT	–	Model-based Testing
MC/DC	–	Modified Condition/Decision Coverage
MDA	–	Model Driven Architecture
MDT	–	Model Driven Testing
MiL	–	Model-in-the-Loop
MiLEST	–	Model-in-the-Loop for Embedded System Test
MIMO	–	Multi-Input-Multi-Output
MISRA™	–	Motor Industry Software Reliability Association
ML	–	MATLAB®
MOF	–	Meta-Object Facility
MOST	–	Media Oriented Systems Transport
MSC	–	Message Sequence Charts

O

OCL	–	Object Constraint Language
OMG	–	Object Management Group

P

PAS	–	Preconditions-Assertions-Synchronization
PCM	–	Powertrain Control Module
PID	–	Proportional-Integral-Derivative controller
PiL	–	Processor-in-the-Loop
PIM	–	Platform-Independent Management
PIT	–	Platform-Independent Testing
pp	–	Partition point
PS	–	Preconditions Synchronization
PSM	–	Platform-Specific Management
PST	–	Platform-Specific Testing

Q

QA	–	Quality Assurance
QVT	–	Query/View/Transformation

R

R	–	Reset signal
R&D	–	Research and Development
RCP	–	Rapid Control Prototyping
RK4	–	Fourth-Order Runge-Kutta Integration Technique

S

SCADE	–	Safety-Critical Application Development Environment
SCB	–	Safety Checker Blockset
SD	–	Sequence Diagram
SE	–	Software Engineering
SF	–	Stateflow®
SigF	–	Signal Feature
SiL	–	Software-in-the-Loop
SISO	–	Single-Input-Single-Output
SL	–	Simulink®

SP	–	Scenario Pattern
STB	–	Safety Test Builder
STFT	–	Short-Time Fourier Transform
SUT	–	System under Test
SW	–	Software

T

T	–	Trigger signal
TA	–	Timed Automata
TAV	–	Testing, Analysis and Verification of Software
TBX	–	Toolbox
TCU	–	Transmission Control Unit
TDD	–	Triggered features identifiable with determinate delay or without a delay
TDGen	–	Test Data Generation in MiLEST
TI	–	Time-independent features identifiable with or without a delay
TID	–	Triggered features identifiable with indeterminate delay
TM	–	Trademark
TMW	–	The MathWorks™, Inc.
TPT	–	Time Partitioning Testing
TSpec	–	Test Specification in MiLEST
TTCN-3	–	Testing and Test Control Notation, version 3
T-VEC	–	Test VECtor

U

U2TP	–	UML 2.0 Testing Profile
UCSD	–	University of California, San Diego
UML®	–	Unified Modeling Language™
USPTO	–	United States Patent and Trademark Office
UTP	–	UML Testing Profile
UTPes	–	UML Testing Profile for Embedded Systems

V

VF	–	Validation Function
VHDL	–	Very High-Speed Integrated Hardware Description Language
VP	–	Verification Pattern

W

WCET	–	Worst-Case Execution Time

Others:

cf.	–	confer
e.g.	–	*Latin: exempli gratia* (for example)
i.e.	–	*Latin: id est* (that is)
w. r. t.	–	with respect to/ with regard to

Bibliography

[ABH^{+}97] Amon, T., Borriello, G., Hu, T., Liu, J.: Symbolic timing verification of timing diagrams using Presburger formulas. In *Proceedings of the 34th Annual Conference on Design Automation*, Pages: 226 – 231, ISBN: 0-89791-920-3. ACM New York, NY, U.S.A., 1997.

[ACK81] Ackrill, J. L.: *Aristotle the philosopher*. ISBN-10: 0192891189. Oxford, 1981.

[AES08] *Automotive Entertainment Systems – 2008 Edition*, Research Report # SC1003, Semicast Research Group, Pages: 202, 2007. http://www.electronics.ca/reports/automotive/entertainment_systems2.html [07/11/08].

[AGH00] Alur, R., Grosu, R., Hur, Y., Kumar, V., Lee, I.: Modular Specification of Hybrid Systems in CHARON. In *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control*. Pittsburgh, 2000.

[AKR^{+}06] Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture – Foundations and Applications*, Volume 4066/2006, Pages: 361 – 375, ISBN: 978-3-540-35909-8, *LNCS.* Springer-Verlag Berlin/Heidelberg, 2006.

[ALE79] Alexander, Ch.: *The Timeless Way of Building*, Oxford University Press, 1979.

[ALL06] Allen, P.: *Service Orientation, winning strategies and best practices*. ISBN: 13-978-0-521-84336-2. Cambridge University Press, 2006.

[ALS^{+}07] Amelunxen, C., Legros, E., Schürr, A., Stürmer, I.: Checking and Enforcement of Modeling Guidelines with Graph Transformations. In *Proceedings of the 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance*, Editors: Schürr, A., Nagl, M., Zündorf, A. 2007.

[AR] Aristotle, *The Politics*, Book IV. http://www.constitution.org/ari/polit_01.htm [04/14/08].

[ART05] *The ARTIST Roadmap for Research and Development. Embedded Systems Design. LNCS*, Volume 3436, Editors: Bouyssounouse, B., Sifakis, J., XV, ISBN: 978-3-540-25107-1. 2005.

[AUFO] Munich University of Technology, Department of Computer Science, *AutoFocus*, research tool for system Modelling, http://autofocus.in.tum.de/ [04/20/2008].

[AUTD] dSPACE GmbH, *AutomationDesk*, commercial tool for testing, http://www.dspace.de/goto?releases [04/20/2008].

[BBH04] Berkenkoetter, K., Bisanz, S., Hannemann, U., Peleska, J.: *HybridUML Profile for UML 2.0*, University of Bremen. 2004.

[BBK98] Broy, M., von der Beeck, M., Krüger, I.: *Softbed: Problemanalyse für ein Großverbundprojekt "Systemtechnik Automobil – Software für eingebettete Systeme"*. Ausarbeitung für das BMBF. 1998 (in German).

[BBN04] Blackburn, M., Busser, R., Nauman, A.: Why Model-Based Test Automation is Different and What You Should Know to Get Started. In *Proceedings of the International Conference on Practical Software Quality*. 2004.

[BBS04] Bienmüller, T., Brockmeyer, U., Sandmann, G.: Automatic Validation of Simulink/Stateflow Models, Formal Verification of Safety-Critical Requirements, Stuttgart. 2004.

[BDG07] Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: *Model-Driven Testing: Using the UML Testing Profile*. ISBN-10: 3540725628, ISBN-13: 978-3540725626. Springer-Verlag, 2007.

[BDH05] Brockmeyer, U., Damm, W., Hungar, H., Josko, B.: Modellbasierte Entwicklung eingebetteter Systeme. In *Proceedings of the Model-Based Development of Embedded Systems*, WS-Nr. 05022, Technischer Bericht, TUBS-SSE-2005-01. 2005 (in German).

[BEI95] Beizer, B.: *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. ISBN-10: 0471120944. John Wiley & Sons, Inc, 1995.

[BEN79] Bennet, S.: *A History of Control Engineering, 1800-1930*. ISBN: 0906048079. Institution of Electrical Engineers, Stevenage, UK. 1979.

[BERT07] Bertolino, A.: Software Testing Research: Achievements, Challenges. *In Proceedings of the International Conference on Software Engineering 2007*, Future of Software Engineering, Pages: 85 – 103, ISBN: 0-7695-2829-5. 2007.

[BFM+05] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A. L., Freund, U., Schlenker, E., Wolff, H.-J.: Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Volume 2, Pages: 1044 – 1049, ISBN: 1530-1591. IEEE Computer Society Washington, DC, 2005.

[BIN99] Binder, R.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. ISBN-10: 0201809389. Addison-Wesley, 1999.

[BJK$^+$05] *Model-Based Testing of Reactive Systems*, Editors: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A., no. 3472. In *LNCS*, Springer-Verlag, 2005.

[BKB05] Bodeveix, J.-P., Koné, O., Bouaziz, R.: Test method for embedded real-time systems. In *Proceedings of the European Workshop on Dependable Software Intensive Embedded Systems*, ERCIM, Pages: 1 – 10. Porto, 2005.

[BKL07] Bräuer, J., Kleinwechter, H., Leicher, A.: µTTCN – an approach to continuous signals in TTCN-3. In *Proceedings of Software Engineering (Workshops)*, Editors: Bleek, W.-G., Schwentner, H., Züllighoven, H., Series LNI, Volume 106, Pages: 55 – 64, ISBN: 978-3-88579-200-0. GI, 2007.

[BKM07] Broy, M., Krüger, I. H., Meisinger, M.: A formal model of services. In *ACM Trans. Softw. Eng. Methodol.* Volume 16, Issue 1, Article 5. 2007.

[BKP$^+$07] Broy, M., Krüger, I. H., Pretschner, A., Salzmann, C.: Engineering Automotive Software. In *Proceedings of the IEEE*, Volume 95, Number 2, Pages: 356 – 373. 2007.

[BM07] Bostroem, P., Morel, L.: *Formal Definition of a Mode-Automata Like Architecture in Simulink/Stateflow*, Turku Centre for Computer Science, Technical Report, No 830, ISBN: 978-952-12-1922-1. Finland, 2007. http://www.tucs.fi/publications/attachment.php?fname=TR830.pdf [07/10/08].

[BMW07] Bostroem, P., Morel, L., Waldén, M.: Stepwise Development of Simulink Models Using the Refinement Calculus Framework, In *Theoretical Aspects of Computing – ICTAC 2007*, Volume 4711/2007, Pages: 79 – 93, ISSN: 0302-9743 1611-3349, ISBN: 978-3-540-75290-5. LNCS, Springer Berlin / Heidelberg 2007.

[BN02] Brökman, B., Notenboom, E.: *Testing Embedded Software*. ISBN: 978-0-3211-5986-1. Addison-Wesley International, 2002.

[BOD05] Bodemann, C. D.: Function-oriented Model-Based Design Development for Real-Time Simulators with MATLAB & Simulink. In *Proceedings of the Model-Based Design Conference*, Munich. 2005.

[BRU87] Brundtland Report, *Our Common Future.* Published by the World Commission on Environment and Development as Annex to General Assembly document A/42/427, Development and International Co-operation: Environment, 1987. http://www.un-documents.net/a42-427.htm [05/23/08]. ISBN-13: 978-0192820808, Oxford: Oxford University Press, 1987.

[BS98] BS 7925-2:1998, *Software testing. Software component testing*, British Standards Institution, ISBN: 0580295567. 1998.

[BSK04] Born, M., Schieferdecker, I., Kath, O. and Hirai, C.: Combining System Development and System Test in a Model-centric Approach. In *Proceedings of the RISE 2004*, Luxembourg. 2004.

[BUR03] Burnstein, I.: *Practical Software Testing*, 1st edition. ISBN-10: 0387951318, ISBN-13: 978-0387951317. Springer-Verlag, 2003.

[CBD$^+$06] Chaparadza, R., Busch, M., Dai, Z. R., Hoffman, A., Lacmene, L., Ngwangwen, T., Ndem, G. C., Serbanescu, D., Schieferdecker, I., Zander-Nowicka, J.: Transformations: UML2 System Models "to" U2TP models, U2TP models "to" TTCN-3 models and, TTCN-3 Code Generation and Execution. In *Proceedings of the ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, Bilbao, Spain. 2006.

[CC00] Copi, I. M., Cohen, C.: *Introduction to logic*, 11th ed. Upper Saddle River, NJ: Prentice-Hall, 2000.

[CC90] Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: a taxonomy, Software. In *IEEE Software*, Volume 7, Issue 1, Pages: 13 – 17, ISSN: 0740-7459. 1990.

[CD06] Conrad, M., Dorr, H.: Model-Based Development of In-Vehicle Software, Design, Automation and Test in Europe. In *Proceedings of the DATE '06*, Volume 1, 6-10, Pages: 1 – 2. 2006.

[CDP$^+$96] Cohen, D. M., Dalal, S. R., Parelius, J., Patton, G. C.: The Combinatorial Design Approach to Automatic Test Generation. In *IEEE Software*, Volume 13, Issue 5, Pages: 83 – 88, ISSN: 0740-7459, IEEE Computer Society Press Los Alamitos, CA, 1996.

[CFB04] Conrad, M., Fey, I., Buhr, K.: Integration of Requirements into Model-Based Development. In *Proceddings of the IEEE Joint Int. Requirements Engineering Conf. 2004: Workshop W-7 on Automotive Requirements Engineering (AuRE'04)*, Pages: 23 – 32, Nagoya, Japan. IEEE, 2004.

[CFG$^+$05] Conrad, M., Fey, I., Grochtmann, M., Klein, T.: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In *Informatik – Forschung und Entwicklung*, Volume 20, Numbers 1-2, ISSN: 0178-3564, 0949-2925. Springer-Verlag Berlin/Heidelberg, 2005 (in German).

[CFS04] Conrad, M., Fey, I., Sadeghipour, S.: Systematic Model-based Testing of Embedded Control Software – The MB^{3}T Approach. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems*, Edinburgh, United Kingdom. 2004.

[CH98] Conrad, M., Hötzer, D.: Selective Integration of Formal Methods in the Development of Electronic Control Units. In *Proceedings of the ICFEM 1998*, 144-Electronic Edition. 1998.

[CHM] Carnegie Mellon University, Department of Electrical and Computer Engineering, *Hybrid System Verification Toolbox for MATLAB – CheckMate*, research tool for system verification, http://www.ece.cmu.edu/~webk/checkmate/ [05/21/2008].

[CLP08] Carter, J. M., Lin, L., Poore, J. H.: Automated Functional Testing of Simulink Control Models. In *Proceedings of the 1st Workshop on Model-based Testing in Practice – MoTip 2008*. Editors: Bauer, T., Eichler, H., Rennoch, A., ISBN: 978-3-8167-7624-6, Berlin, Germany. Fraunhofer IRB Verlag, 2008.

[CON04a] Conrad, M.: *Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien.* PhD thesis. Deutscher Universitätsverlag, Wiesbaden (D), 2004 (in German).

[CON04b] Conrad, M.: *A Systematic Approach to Testing Automotive Control Software*, Detroit U.S.A., SAE Technical Paper Series, 2004-21-0039. 2004.

[CON08] Continental Automotive, *Adaptive Cruise Control – Chassis Electronics Combined with Safety Aspects*, ©Continental Teves AG & Co. oHG 2008, http://www.conti-online.com/generator/www/de/en/cas/cas/themes/products/electronic_brake_and_safety_systems/driver_assistance_systems/acc_today_en.html [05/18/08].

[COP79] Copi, I. M.: *Symbolic Logic* (5th edition), Macmillan Coll Div, ISBN-10: 0023249803, ISBN-13: 9780023249808, April 1979.

[CS03] Caplat, G., Sourrouille, J. L. S.: Considerations about Model Mapping. In *Proceedings of the 4th Workshop in Software Model Engineering – in conjunction with the Sixth International Conference on the Unified Modelling Language*, San Francisco, U.S.A. 2003. http://www.metamodel.com/wisme-2003/18.pdf [05/09/08].

[CTE] Razorcat Development GmbH, *Classification Tree Editor for Embedded Systems – CTE/ES*, commercial tool for testing, http://www.razorcatdevelopment.de/ [04/20/08].

[DAG] Daimler AG, http://www.daimler.com/ [04/22/08].

[DAI04] Dai, Z. R.: Model-Driven Testing with UML 2.0. In *Proceedings of the 2nd European Workshop on Model Driven Architecture (EWMDA)*, Canterbury, England. 2004.

[DAI06] Dai, Z. R.: *An Approach to Model-Driven Testing with UML 2.0, U2TP and TTCN-3*. PhD thesis, Technical University Berlin, ISBN: 978-3-8167-7237-8. Fraunhofer IRB Verlag, 2006.

[DCB04] Dajani-Brown, S., Cofer, D., Bouali, A.: Formal Verification of an Avionics Sensor Voter Using SCADE. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Volume 3253/2004, *LNCS*, Pages: 5 – 20, ISBN: 978-3-540-23167-7, ISSN: 0302-9743 1611-3349. Springer-Verlag Berlin/Heidelberg, 2004.

[DESS01] DESS – Software Development Process for Real-Time Embedded Software Systems, *The DESS Methodology*. Deliverable D.1, Version 01 – Public, Editors: van Baelen, S., Gorinsek, J., Wills, A. 2001.

[DF03] Dulz, W., Fenhua, Z.: MaTeLo – Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3. In *Proceedings of the 3rd International Conference on Quality Software*, Page: 336, ISBN: 0-7695-2015-4. IEEE Computer Society Washington, DC, U.S.A., 2003.

[DFG01] *Sequential Monte Carlo methods in practice*. Editors: Doucet, A., de Freitas N., Gordon, N., ISBN: 0-387-95146-6. Springer-Verlag, New York, 2001.

[DGN04] Dai Z. R., Grabowski J., Neukirchen H., Pals H. From Design to Test – Applied to a Roaming Algorithm for Bluetooth Devices. Next Generation Testing for Next Generation Networks. In *Proceedings of the TestCom 2004,* St Anne's College, Oxford, UK. Springer-Verlag, 2004.

[DIJ72] Dijkstra, E. W.: Notes on Structured Programming. In *Structured Programming*, Volume 8 of *A.P.I.C. Studies in Data Processing*, Part 1, Editor: Hoare C. A. R., Pages: 1 – 82. Academic Press, London/New York, 1972.

[DIN08] Din, G.: *A Performance Test Design Method and its Implementation Patterns for Multi-Services Systems.* PhD thesis, Technical University Berlin. 2008.

[DM_D07] D-Mint Consortium. Deployment of model-based technologies to industrial testing, *Milestone 1, Deliverable 2.1 – Test Modeling, Test Generation and Test Execution with Model-based Testing.* 2007.

[D-Mint08] D-Mint Project – *Deployment of model-based technologies to industrial testing.* 2008. http://d-mint.org/ [05/09/08].

[DP80] Dormand Jr., Prince P.: A family of embedded Runge-Kutta formulae. In *Journal of Computational and Applied Mathematics*, Pages: 19 – 26. 1980.

[DS02] Dempster, D., Stuart, M.: *Verification methodology manual, Techniques for Verifying HDL Designs*, ISBN: 0-9538-4822-1. Teamwork International, June 2002.

[DSP] dSPACE GmbH, http://www.dspace.de/ww/en/gmb/home.cfm [04/22/08].

[DSW+03] Damm, W., Schulte, C., Wittke, H., Segelken, M., Higgen, U., Eckrich, M.: Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. Pages: 340 – 344. GI Jahrestagung (1) 2003 (in German).

[EKM+07] Ermagan, V., Krüger, I., Menarini, M., Mizutani, J.-I., Oguchi, K., May, W.D.: Towards Model-Based Failure-Management for Automotive Software. In *Proceedings of the ICSE 4th International Workshop on Software Engineering for Automotive Systems (SEAS'07)*, p. 8, Minneapolis, MN, U.S.A. IEEE Computer Society, 2007.

[EKM06] Ermagan, V., Krüger, I., Menarini, M.: Model-Based Failure Management for Distributed Reactive Systems. In *Proceedings of the 13th Monterey Workshop, Composition of Embedded Systems, Scientific and Industrial Issues*, Editors: Kordon, F, Sokolsky, O., *LNCS*, Paris, France. Springer-Verlag, 2006.

[EMBV] OSC – Embedded Systems AG, Embedded*Validator*, commercial verification tool, http://www.osc-es.de [04/20/08].

[EMC+99] Ehring, H., Mahr, B., Cornelius, F., Große-Rhode, M., Zeitz, P.: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, Barcelona, Berlin, Heidelberg, Hongkong, London, Mailand, New York, Paris, Singapur, Tokio, 1999 (in German).

[ENC03] Encontre, V.: *Testing embedded systems: Do you have the GuTs for it?*. IBM, 2003. http://www-128.ibm.com/developerworks/rational/library/459.html [04/18/2008].

[ERK04] Erkkinen, T.: Production Code Generation for Safety-Critical Systems. SAE 2004 World Congress & Exhibition. In *SP-1822 – Planned by Diesel Engine Committee/ Powerplant Activity Governing Board*. 2004.

[ETSI07] ETSI European Standard (ES) 201 873-1 V3.2.1 (2007-02): *The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. European Telecommunications Standards Institute, Sophia-Antipolis, France. 2007.

[EU06] European Commission Report on the Public Consultation, Review of the EU strategy to reduce CO2 emissions and improve fuel efficiency from cars. 2006. http://ec.europa.eu/environment/air/transport/co2/pdf/public_consultation_report.pdf [05/25/2008].

[FDI+04] Fokking, W. J., Deussen, P. H., Ioustinova, N., Seubers, J., van de Pol J.: *Towards model-based test generation and validation for TTCN-3*, Milestone 1, Deliverable 1.2.1, Tests and Testing Methodologies with Advanced Languages (TT-Medal). 2004. http://www.tt-medal.org/results/download/work1/D.1.2.1.v1.0.FINAL.TowardsModelBasedTestGenerationAndValidationForTTCN-3_PUB.pdf [05/12/08].

[FG07] Farkas, T., Grund, D.: Rule Checking within the Model-Based Development of Safety-Critical Systems and Embedded Automotive Software. In *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS'07)*, Pages: 287 – 294. 2007.

[FRA05] Franzen T.: *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. Pages: 172, ISBN-10: 1568812388, ISBN-13: 978-1568812380. A K Peters, Ltd., 2005.

[FS08] Frost and Sullivan, *Strategic Analysis of the European Automotive Advanced Electronic Controller Markets*, M1C6-18. March 2008.

[GCF06] Großmann, J., Conrad, M., Fey, I., Krupp, A., Lamberg, K., Wewetzer, C.: TestML – A Test Exchange Language for Model-based Testing of Embedded Software. In *Proceedings of Automotive Workshop San Diego (ASWWSD2006)*. 2006.

[GG93] Grochtmann, M., Grimm, K.: Classification Trees for Partition Testing. In *Software Testing, Verification & Reliability*, Volume 3, Number 2, Pages: 63 – 82. Wiley, 1993.

[GHS+07] Gehrke, M., Hirsch, M., Schäfer, W., Niggemann, O., Stichling, D., Nickel, U.: Typisierung und Verifikation zeitlicher Anforderungen automotiver Software Systeme. In *Proceedings of Model Based Engineering of Embedded Systems III*, Editors: Conrad, M., Giese, H., Rumpe, B., Schätz, B.: TU Braunschweig Report TUBS-SSE 2007-01, 2007 (in German).

[GKS99] Grosu, R., Krüger, I., Stauner, T.: *Hybrid Sequence Charts*. Technical Report TUM-I9914, Technische Universität München, 1999.

[GMS07] Gadkari, A. A., Mohalik, S., Shashidhar, K. C., Yeolekar, A., Suresh, J., Ramesh, S.: Automatic Generation of Test-Cases Using Model Checking for SL/SF Models. In *Proceedings of MoDeVVa'07 in conjunction with MoDELS2007.* Nashville, Tenesee. 2007. http://www.modeva.org/2007/modevva07.pdf [05/09/08].

[GOA05] Grindal, M., Offutt, J., Andler, S. F.: Combination testing strategies: a survey. In *Software Testing, Verification and Reliability*. Volume 15, Issue 3, Pages: 167 – 199. John Wiley & Sons, Ltd., 2005.

[GR06] Guerrouat, A., Richter, H.: A component-based specification approach for embedded systems using FDTs. In *Proceedings of Specification and Verification of Component-Based Systems Workshop (SAVCBS 2005)*, *ACM SIGSOFT Software Engineering Notes*, Volume 31, Issue 2 (March 2006), Article No. 14, 2006, ISSN: 0163-5948, ACM New York, NY, U.S.A. 2006.

[GRI03] Grimm, K.: Software technology in an automotive company: major challenges. In *Proceedings of the 25th International Conference on Software Engineering*, ISSN: 0270-5257, 0-7695-1877-X, Portland, Oregon, U.S.A., IEEE Computer Society Washington, DC, 2003.

[GRI95] Grimm, K.: *Systematisches Testen von Software. Eine neue Methode und eine effektive Teststrategie*. PhD thesis, Technical University Berlin, GMD-Bericht, 251, ISBN: 3-486-23547-8. GMD Forschungszentrum Informationstechnik, 1995 (in German).

[GSW08] Großmann, J., Schieferdecker, I., Wiesbrock, H. W.: Modeling Property Based Stream Templates with TTCN-3. In *Proceedings of the IFIP 20th Intern. Conf. on Testing Communicating Systems (TestCom 2008)*, Tokyo, Japan. 2008.

[GUT99] Gutjahr, W. J.: Partition testing vs. random testing: the influence of uncertainty. In *IEEE Transactions on Software Engineering*, Volume 25, Issue 5, Pages: 661 – 674, ISSN: 0098-5589. IEEE Press Piscataway, NJ, 1999.

[GW07] Gips C., Wiesbrock H.-W. Notation und Verfahren zur automatischen Überprüfung von temporalen Signalabhängigkeiten und -merkmalen für modellbasiert entwickelte Software. In *Proceedings of Model Based Engineering of Embedded Systems III*, Editors: Conrad, M., Giese, H., Rumpe, B., Schätz, B.: TU Braunschweig Report TUBS-SSE 2007-01, 2007 (in German).

[HEL$^+$05] Helmerich, A., Koch, N. and Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., Wild, T. Wimmel, G.: *Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area,* Final Report for the European Commission, Brussels Belgium, 2005.

[HEN00] Henzinger, T. A.: The theory of hybrid automata. *Proceedings of the 11th Annual Symposium on Logic in Computer Science* (LICS), IEEE Computer Society Press, 1996, Pages: 278 – 292. An extended version appeared in *Verification of Digital and Hybrid Systems*, Editors: Inan, M. K., Kurshan, R. P., NATO ASI Series F: Computer and Systems Sciences, Volume 170, Pages: 265 – 292. Springer-Verlag, 2000.

[HET98] Hetzel, W. C.: *The Complete Guide to Software Testing*. Second edition, ISBN: 0-89435-242-3. QED Information Services, Inc, 1988.

[HP85] Harel, D., Pnueli, A.: On the Development of Reactive Systems. In K. R. Apt, *Logics and Models of Concurrent Systems*, NATO, ASI Series, Band 13, S. 447-498. New York, ISBN: 0-387-15181-8. Springer-Verlag, 1985.

[IEC05] Functional safety and IEC 61508, 2005.
http://www.iec.ch/zone/fsafety/fsafety_entry.htm [05/09/08].

[IEG90] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std. 610.12-1990. E-ISBN: 0-7381-0391-8. 1990. http://www.scribd.com/doc/2893755/IEEE-610121990-IEEE-Standard-Glossary-of-Software-Engineering-Terminology [05/14/08].

[ISO_FS] ISO/NP PAS 26262, Road vehicles - Functional safety, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43464 [05/09/08].

[ISO04] ISO/IEC Standard No. 9126: Software engineering – Product quality; Parts 1–4. Geneva, Switzerland, 2001-2004.

[ISTQB06] International Software Testing Qualification Board. *Standard glossary of terms used in Software Testing*. Version 1.2, Produced by the ‘Glossary Working Party’, Editor: van Veenendaal E. 2006.

[ITU96] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.

[ITU99] ITU-TS. Recommendation Z.120 (11/99): MSC 2000. Geneva, 1999.

[JJR05] Jeannet, B., Jéron, Rusu, V., Zinovieva, E.: Symbolic Test Selection Based on Approximate Analysis. In *Proceedings of TACAS 2005*. Editors: Halbwachs, N., Zuck, L., *LNCS 3440*, Pages: 349 – 364, Berlin, Heidelberg. Springer-Verlag, 2005.

[JUMB] Software Quality Research Laboratory, *Java Usage Model Builder Library – JUMBL*, research model-based testing prototype, http://www.cs.utk.edu/sqrl/esp/jumbl.html [04/20/08].

[KH96] Kamen, E. W., Heck, B. S.: *Fundamentals of Signals and Systems Using MATLAB*. ISBN-10: 0023619422, ISBN-13: 978-0023619427. Prentice Hall, 1996.

[KHJ07] Kamga, J., Herrmann, J., Joshi, P.: Deliverable: D-MINT automotive case study - Daimler, Deliverable 1.1, *Deployment of model-based technologies to industrial testing*, ITEA2 Project, 2007.

[KIL05] Kilian, K.: *Modern Control Technology*. Thompson Delmar Learning. ISBN: 1-4018-5806-6. 2005.

[KLP+04] Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*. ISSN: 1071-9458, ISBN: 0-7695-2215-7, Pages: 139 – 150. IEEE Computer Society Washington, DC, 2004.

[KM08] Kröger, F., Merz, S.: *Temporal Logic and State Systems Series: Texts in Theoretical Computer Science*. 436 pages, ISBN: 978-3-540-67401-6. An EATCS Series, 2008.

[KRI05] Krishnan, R.: *Future of Embedded Systems Technology*, Research Report #G229R, Market Study, Page: 354. BCCresearch, June 2005.

[KRÜ00] Krüger, I. H.: *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[KRÜ05] Krüger, I. H.: Service-oriented software and systems engineering - a vision for the automotive domain. In *Proceedings Formal Methods and Models for Co-Design. (MEMOCODE '05). The third ACM and IEEE International Conference*, Page: 150. 2005.

[KUO03] Kuo, B. C.: *Automatic Control Systems*, ISBN-10: 0471381489, ISBN-13: 978-0471381488. Wiley & Sons, 2003.

[LABV] LabView, National Instruments, http://www.ni.com/labview/ [04/15/08].

[LBE+04] Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., Fey, I.: Model-based testing of embedded automotive software using MTest. In *Proceedings of SAE World Congress*, Detroit, US, 2004.

[LEH03] Lehmann, E. (then Bringmann, E.): *Time Partition Testing, Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*, PhD thesis, Technical University Berlin, 2003 (in German).

[LIU08] Liu, T.: *Development of UML Test Profile for Embedded Systems*, Diploma Thesis, Technical University Berlin, *to be submitted* in October 2008.

[LK08] Lehmann, E., Krämer, A.: Model-based Testing of Automotive Systems. In *Proceedings of IEEE ICST 08*, Lillehammer, Norway. 2008.

[LKK$^+$06] Lehmann, E., Krämer, A., Lang, T., Weiss, S., Klaproth, Ch., Ruhe, J., Ziech, Ch.: *Time Partition Testing Manual*, Version 2.4, 2006.

[LL90] Le Lann, G.: Critical issues for the development of distributed real-time computing systems, G.; Distributed Computing Systems. In the *Proceedings of the Second IEEE Workshop on Future Trends of*, Pages: 96 – 105, ISBN: 0-8186-2088-9, Cairo. 1990.

[LN05] Lee, E. A., Neuendorffer, S.: Concurrent models of computation for embedded software. In *IEE Proceedings – Computers and Digital Technologies*, Volume 152, Issue 2, Pages: 239 – 250, ISSN: 1350-2387. 2005.

[LV04] Lazić, Lj., Velašević, D.: Applying simulation and design of experiments to the embedded software testing process. In *Software Testing, Verification & Reliability*, Volume 14, Issue 4, Pages: 257 – 282, ISSN: 0960-0833. John Wiley and Sons Ltd. Chichester, UK, UK, 2004.

[LW00] Lehmann, E., Wegener, J.: Test Case Design by Means of the CTE XL. In *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, Kopenhagen, Denmark. 2000.

[LY94] Lee, T., Yannakakis, M.: Testing Finite-State Machines: State Identification and Verification. In *IEEE Transactions on Computers*, Volume 43, Issue 3, Pages: 306 – 320, ISSN: 0018-9340. IEEE Computer Society Washington, DC, 1994.

[LZ05] Lee, E. A., Zheng, H.: Operational Semantics of Hybrid Systems. In *Proceedings of Hybrid Systems: Computation and Control: 8th International Workshop*, HSCC, *LNCS 3414*, Zurich, Switzerland. 2005.

[MA00] Marre, B., Arnould, A.: Test sequences generation from LUSTRE descriptions: GATEL. In *Proceedings of ASE of the 15th IEEE International Conference on Automated Software Engineering*, Pages: 229 – 237, ISBN: 0-7695-0710-7, Grenoble, France. IEEE Computer Society Washington, DC, 2000.

[MAN85] Mandl, R.: Orthogonal Latin Squares: An application of experiment design to compiler testing. In *Communications of the ACM*, Volume 28, Issue 10, Pages: 1054 – 1058, ISSN: 0001-0782. ACM New York, NY, U.S.A., 1985.

[MATHML] The MathWorks™, Inc., *MATLAB*®, http://www.mathworks.com/products/matlab/ http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html [05/09/08].

[MATHSF] The MathWorks™, Inc., *Stateflow®*, http://www.mathworks.com/products/stateflow/ [05/09/08], http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/ [05/09/08].

[MATHSL] The MathWorks™, Inc., *Simulink®*, http://www.mathworks.com/products/simulink/ [05/09/08], http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/ [05/09/08].

[MATL] All4Tec, *Markov Test Logic – MaTeLo*, commercial model-based testing tool, http://www.all4tec.net/ [04/20/08].

[MATT] The University of Montana, *MATLAB Automated Testing Tool – MATT*, research model-based testing prototype, http://www.cs.umt.edu/RTSL/matt/ [04/20/08].

[MBG] MBTech Group, http://www.mbtech-group.com/cz/electronics_solutions/test_engineering/provetechta_overview.html [04/22/08].

[MDA] OMG: MDA Guide V1.0.1, June, 2003. http://www.omg.org/docs/omg/03-06-01.pdf [05/09/08].

[MEN97] Mendelson, E.: *Introduction to Mathematical Logic*, 4th Edition, ISBN-10: 412808307, ISBN-13: 978-0412808302. Chapman & Hall/CRC, 1997.

[MEVAL] IT Power Consultants, *MEval*, commercial tool for testing, http://www.itpower.de/meval.html [04/20/2008].

[MIS] The Motor Industry Software Reliability Association (MISRA), http://www.misra.org.uk/ [05/26/2008].

[MLN03] Mikucionis, M., Larsen, K. G, Nielsen, B.: *Online On-the-Fly Testing of Real-time Systems*, ISSN 0909-0878. BRICS Report Series, RS-03-49, Denmark, 2003.

[MOD] Modelica Association, http://www.modelica.org/ [04/15/08].

[MOF] OMG: Meta-Object Facility (MOF), version 1.4, http://www.omg.org/technology/documents/formal/mof.htm [05/09/08].

[MOS97] Mosterman, P. J.: *Hybrid dynamic systems: a hybrid bond graph modeling paradigm and its application in diagnosis*, PhD thesis, Faculty of the Graduate School of Vanderbilt University, Electrical Engineering. 1997.

[MP07] Marrero Pérez, A.: *Simulink Test Design for Hybrid Embedded Systems*, Diploma Thesis, Technical University Berlin, January 2007.

[MRG] The MathWorks™, Inc., *MATLAB® Report Generator™*, http://www.mathworks.com/products/ML_reportgenerator/ [05/11/08].

[MSF05] Mann, H., Schiffelgen, H., Froriep, R.: *Einführung in die Regelungstechnik, Analoge und digitale Regelung, Fuzzy-Regler, Regel-Realisierung, Software*. ISBN-10: 3-446-40303-5. Hanser Fachbuchverlag, 2005 (in German).

[MTEST] dSPACE GmbH, *MTest*, commercial MBT tool, http://www.dspaceinc.com/ww/en/inc/home/products/sw/expsoft/mtest.cfm [04/20/2008].

[MW04] Maxton, G. P., Wormald, J.: *Time for a Model Change: Re-engineering the Global Automotive Industry*, ISBN: 978-0521837156. Cambridge University Press, 2004.

[MW91] Marzullo, K., Wood, M.: Making real-time reactive systems reliable. In *ACM SIGOPS Operating Systems Review*, Volume 25, Issue 1 (January 1991), Pages: 45 – 48, ISSN: 0163-5980, New York, U.S.A. ACM Press New York, 1991.

[MYE79] Myers, G. J.: *The Art of Software Testing*. ISBN-10: 0471043281. John Wiley & Sons, 1979.

[NEU04] Neukirchen, H. W.: *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*, PhD thesis, Georg-August-Universiät zu Göttingen, 2004. http://webdoc.sub.gwdg.de/diss/2004/ neukirchen/index.html [05/09/08].

[NK04] Neema, S., Karsai, G.: *Embedded Control Systems Language for Distributed Processing (ECSL-DP)*, Technical report, ISIS-04-505, Vanderbilt University, 2003-2004.

[NLL03] Nicol, D. M., Liu, J., Liljenstam, M., Guanhua, Y.: Simulation of large scale networks using SSF. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, Volume 1, Pages: 650 – 657, Vol.1, New Orleans, Louisiana. ACM, 2003.

[NM07] Nickovic, D., Maler, O.: AMT: A Property-based Monitoring Tool for Analog Systems. In *Proceedings of the Formal Modelling and Analysis of Timed Systems (FORMATS)*, Volume 4763/2007, ISSN: 0302-9743, 1611-3349, ISBN: 978-3-540-75453-4, Pages: 304 – 319. Springer-Verlag Berlin/Heidelberg, 2007.

[NSK03] Neema, S., Sztipanovits, J., Karsai, G.: Constraint-Based Design-Space Exploration and Model, Synthesis. In *Proceedings of EMSOFT 2003*, Pages: 290 – 305, *LNCS* 2855, 2003.

[OECD05] *OECD Science, Technology and Industry: Scoreboard 2005*. Volume 2005, Issue 30, Complete Edition – ISBN: 9264010556, Industry, Services & Trade, Pages: 1 – 214. 2005.

[OECD08] *OECD Factbook 2008: Economic, Environmental and Social Statistics*, ISBN: 92-64-04054-4, © OECD 2008.

[OLE07] Olen, M.: *The New Wave in Functional Verification: Algorithmic Testbench Technology*, white paper, Mentor Graphics Corporation. 2007. http://www.edadesignline.com/showArticle.jhtml?articleID=197800043 [08/08/08].

[PFT+92] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T.: "Runge-Kutta Method" and "Adaptive Step Size Control for Runge-Kutta." §16.1 and 16.2 in *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed.*, Pages: 704 – 716, Cambridge, England: Cambridge University Press, 1992.

[PHPS03] Philipps, J., Hahn, G., Pretschner, A., Stauner, T.: Prototype-based tests for hybrid reactive systems. In *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, Pages: 78 – 84, ISSN: 1074-6005, ISBN: 0-7695-1943-1. IEEE Computer Society, Washington, DC, U.S.A., 2003.

[POR96] Porat, B.: *A Course in Digital Signal Processing*, 632 pages, ISBN-10: 0471149616, ISBN-13: 978-0471149613. Wiley, 1996.

[PPW+05] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, U.S.A., Pages: 392 – 401, ISBN: 1-59593-963-2. ACM New York, NY, USA, 2005.

[PRE03] Pretschner, A.: Compositional generation of MC/DC integration test suites. In *Proceedings TACoS'03*, Pages: 1 – 11. *Electronic Notes in Theoretical Computer Science* 6, 2003. http://citeseer.ist.psu.edu/633586.html [05/09/08].

[PRE03b] Pretschner, A.: *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis. Technical University Munich, 2003 (in German).

[PRE04] Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: Model Based Testing for Real – The Inhouse Card Case Study. In *International Journal on Software Tools for Technology Transfer*. Volume 5, Pages: 140 – 157. Springer-Verlag, 2004.

[PRO03] Prowell, S. J.: JUMBL: A Tool for Model-Based Statistical Testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, Volume 9, ISBN: 0-7695-1874-5. IEEE Computer Society Washington, DC, 2003.

[PTD05] ETSI Draft Technical Report DTR/MTS-00091, v.1.2.1. *Methods for Testing and Specification (MTS); Patterns in Test Development (PTD)*, European Telecommunications Standards Institute, Sophia-Antipolis, France. 2005.

[RAU02] Rau, A.: *Model-Based Development of Embedded Automotive Control Systems*, PhD thesis, University of Tübingen, 2002.

[REACTT] Reactive Systems, Inc., *Reactis Tester*, commercial model-based testing tool, http://www.reactive-systems.com/tester.msp [04/20/08].

[REACTV] Reactive Systems, Inc., *Reactis Validator*, commercial validation and verification tool, http://www.reactive-systems.com/reactis/doc/user/user009.html, http://www.reactive-systems.com/validator.msp [07/03/08].

[ROT98] Richardson, D, O'Malley, O., Tittle, C.: Approaches to specification-based testing. In *Proceedings of ACM SIGSOFT Software Engineering Notes*, Volume 14, Issue 8, Pages: 86 – 96, ISSN: 0163-5948. ACM New York, NY, 1998.

[RT92] RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*. Washington, D.C., Radio Technical Commission for Aeronautics (RTCA, Inc.), 1992. http://www.rtca.org/ [05/09/08].

[SBG06] Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: Testing of Embedded Control Systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, International Conference on Software Engineering, ISBN: 1-59593-402-2, Shanghai, China. ACM New York Press, 2006.

[SCB] TNI-Software, *Safety Checker Blockset*, commercial model-based testing tool, http://www.tni-software.com/en/produits/safetychecker blockset/index.php [04/20/08].

[SCD$^+$07] Stürmer, I., Conrad, M., Dörr, H., Pepper P.; Systematic Testing of Model-Based Code Generators. In *IEEE Transactions on Software Engineering*. Volume 33, Issue 9, Pages: 622 – 634, ISSN: 0098-5589. IEEE Press Piscataway, NJ, 2007.

[SCH06] Schmid, M.: Automotive Bus Systems. In *Atmel Applications Journal.* Automotive Applications, Volume 6. ATMEL® Applications Journal Winter, 2006.

[SD07] Sims S., DuVarney D. C.: Experience Report: the Reactis Validation Tool. In *Proceedings of the ICFP '07 Conference*, Volume 42, Issue 9, Pages: 137 – 140, ISSN: 0362-1340. ACM New York, NY, U.S.A., 2007.

[SDG$^+$07] Stürmer, I., Dörr, H., Giese, H., Kelter, U., Schürr, A., Zündorf, A.: Das MATE Projekt - visuelle Spezifikation von MATLAB-Analysen und Transformationen. In *Proceedings of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES)*, Pages: 73 – 82, Editors: Conrad, M., Giese, H., Rumpe, B., Schätz, B., Informatik-Bericht, Number 2007-1, Schloss Dagstuhl, Germany. Technische Universität Braunschweig, 2007 (in German).

[SG03] Spencer, R. R., Ghausi, M. S.: *Introduction to electronic circuit design*. ISBN: 0201361833 9780201361834. Upper Saddle River, N.J.: Prentice Hall/Pearson Education, Inc., 2003.

[SG07] Schieferdecker, I., Großmann, J.: Testing Embedded Control Systems with TTCN-3. In *Proceedings Software Technologies for Embedded and Ubiquitous Systems SEUS 2007*, Pages: 125 – 136, *LNCS* 4761, ISSN: 0302-9743, 1611-3349, ISBN: 978-3-540-75663-7 Santorini Island, Greece. Springer-Verlag Berlin/Heidelberg, 2007.

[SL05] Spillner, A., Linz, T.: *Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified Tester Foundation Level nach ASQF- und ISTQB-Standard*. ISBN: 3-89864-358-1. dpunkt.Verlag GmbH Heidelberg, 2005 (in German).

[SLDV] The MathWorks™, Inc., *Simulink® Design Verifier*™, commercial model-based testing tool, http://www.mathworks.com/products/sldesignverifier [04/20/2008].

[SLVV] The MathWorks™, Inc., *Simulink® Verification and Validation*™, commercial model-based verification and validation tool, http://www.mathworks.com/products/simverification/ [04/20/2008].

[SM01] Sax, E., Müller-Glaser, K.-D.; A Seamless, Model-based Design Flow for Embedded Systems in Automotive Applications. In *Proceedings of the 1st International Symposium on Automotive Control*, Shanghai, China, 2001.

[SOE00] Soejima S.: Examples of usage and spread of Dymola within Toyota. In *Proceedings of Modelica Workshop 2000*, Pages: 55 – 60, Lund, Sweden. 2000.

[SRG] The MathWorks™, Inc., *Simulink® Report Generator*™, http://www.mathworks.com/products/SL_reportgenerator/ [05/11/08].

[SRK$^+$00] Silva, B. I., Richeson K., Krogh B., Chutinan A.: Modeling and verifying hybrid dynamic systems using CheckMate. In *Proceedings of the 4th International Conference on Automation of Mixed Processes (ADPM 2000)*, Pages: 237 – 242. 2000.

[STB] TNI-Software, *Safety Test Builder*, commercial model-based testing tool, http://www.tni-software.com/en/produits/safetytestbuilder/index.php [04/20/08].

[STEST] The MathWorks™, Inc., *SystemTest*™, commercial tool for testing, http://www.mathworks.com/products/systemtest/ [04/20/2008].

[SYN05] SynaptiCAD News: SynaptiCAD and Actel Press Release, January 2005, http://www.syncad.com/pr_wl_rtb_actel_2005.htm?%20SynaptiCADSessionID=ab62c5f8b4595890f2 [05/08/08].

[SZ06] Schäuffele, J., Zurawka, T.: *Automotive Software Engineering*, ISBN: 3528110406. Vieweg, 2006.

[TB04] Torrisi, F. D., Bemporad, A.: HYSDEL – a tool for generating computational hybrid models for analysis and synthesis problems. In *IEEE Transactions on Control Systems Technology*, Volume: 12, Issue: 2, Pages: 235 – 249, ISSN: 1558-0865. IEEE, 2004.

[TELD] Telelogic® AB, Telelogic DOORS®, http://www.telelogic.com/products/doors/index.cfm [07/03/08].

[TEM08] TEMEA Project – Testspezifikationstechnologie und -methodik für eingebettete Echtzeitsysteme im Automobil, http://www.temea.org/ [05/11/08] (in German).

[TIW02] Tiwari, A.: *Formal semantics and analysis methods for Simulink Stateflow models*. SRI International. Technical report, 2002. http://www.csl.sri.com/~tiwari/stateflow.html [05/09/08].

[TPT] PikeTec, *Time Partitioning Testing – TPT*, commercial model-based testing tool, http://www.piketec.com/products/tpt.php [04/20/2008].

[TUN04] Tung, J.: From Specification and Design to Implementation and Test: A Platform for Embedded System Development, The MathWorks Automotive Workshop, 2004.

[TVEC] T-VEC Technologies, Inc., *Test VECtor Tester for Simulink – T-VEC*, commercial model-based testing tool, http://www.t-vec.com/solutions/simulink.php [07/03/08].

[TYZ+03] Tsai, W. T., Yu, L., Zhu, F., Paul, R.: Rapid Verification of Embedded Systems Using Patterns. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, Pages: 466 – 471, ISBN: 0-7695-2020-0, 2003.

[TYZ05] Tsai, W.-T., Yu, L., Zhu, F., Paul, R.: Rapid embedded system testing using verification patterns. In *IEEE Software*, Volume 22, Issue 4, Pages: 68 – 75, ISSN: 0740-7459, Los Alamitos, CA, USA. IEEE Computer Society Press, 2005.

[UL06] Utting M., Legeard B. *Practical Model-Based Testing: A Tools Approach.* ISBN-13: 9780123725011. Elsevier Science & Technology Books, 2006.

[UML] OMG: UML 2.0 Superstructure Final Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02 [05/09/08].

[UPL06] Utting M., Pretschner A., Legeard B. *A taxonomy of model-based testing*, ISSN: 1170-487X, 2006.

[UTP] OMG: UML 2.0 Testing Profile. Version 1.0 formal/05-07-07. Object Management Group, 2005.

[UTT05] Utting M. Model-Based Testing. In *Proceedings of the Workshop on Verified Software: Theory, Tools, and Experiments VSTTE 2005*. 2005.

[VECI] Vector Informatik GmbH, http://www.vector-worldwide.com/ [04/22/08].

[VM06] V-Modell® XT, version 1.2.1, 2006, ftp://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.2.1/Documentation/V-Modell-XT-Complete.pdf [07/05/08].

[VS04] Vouffo-Feudjio, A., Schieferdecker, I.: Test Patterns with TTCN-3. In *Proceedings Formal Approaches to Software Testing, 4th International Workshop (FATES 2004)*, Pages: 170 – 179, ISSN: 0302-9743, ISBN: 978-3-540-25109-5, Volume 3395 *LNCS*, Linz, Austria. Springer-Verlag, 2005.

[VS06] Vega D.-E., Schieferdecker I., Din G.. Towards Quality of TTCN-3 Tests. In *Proceedings of SAM'06: Fifth Workshop on System Analysis and Modeling*, May 31–June 2, University of Kaiserslautern, Germany, 2006.

[VSD07] Vega, D., Schieferdecker, I., Din, G.: Test Data Variance as a Test Quality Measure: Exemplified for TTCN-3. In *Proceedings Testing of Software and Communicating Systems 2007*, Volume 4581, Pages: 351 – 364, ISBN: 978-3-540-73065-1, ISSN: 0302-9743, 1611-3349. Springer-Verlag Berlin/Heidelberg, 2007.

[WAL01] Wallmüller E. *Software- Qualitätsmanagement in der Praxis*. ISBN-10: 3446213678. Hanser Verlag, 2001 (in German).

[WCF02] Wiesbrock, H.-W., Conrad M., Fey, I.: Pohlheim, Ein neues automatisiertes Auswerteverfahren für Regressions und Back-to-Back-Tests eingebetteter Regelsysteme, In *Softwaretechnik-Trends*, Volume 22, Issue 3, Pages: 22 – 27. 2002 (in German).

[WEY88] Weyuker, E.: The Evaluation of Program-based Software Test Data Adequacy Criteria. In *Communications of the ACM*, Volume 31, Issue 6, Pages: 668 – 675, ISSN: 0001-0782. ACM New York, 1988.

[WG07] Wiesbrock, H.-W. Gips, C.: *Konzeption eines Tools zur automatischen Testauswertung von Hard- und Softwaretests*. Unpublished. 2007 (in German).

[WTB] SynaptiCAD, *Waveformer Lite 9.9 Test-Bench* with Reactive Test Bench, commercial tool for testing, http://www.actel.com/documents/reactive_tb_tutorial.pdf [04/20/08].

[WW06] Wappler, S., Wegener, J.: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Pages: 1925 – 1932, ISBN: 1-59593-186-4 Seattle. Washington, U.S.A. ACM, 2006.

[XIO08] Xiong, X.: *Systematic Test Data Generation for Embedded Systems*, Diploma Thesis, Technical University Berlin, February 2008.

[ZAN07] Zander-Nowicka, J.: Reactive Testing and Test Control of Hybrid Embedded Software. In *Proceedings of the 5th Workshop on System Testing and Validation (STV 2007)*, in conjunction with ICSSEA 2007, Editors: Garbajosa, J., Boegh, J., Rodriguez-Dapena, P., Rennoch, A., Pages: 45 – 62, ISBN: 978-3-8167-7475-4, Paris, France. Fraunhofer IRB Verlag, 2007.

[ZDS$^+$05] Zander, J., Dai, Z. R., Schieferdecker, I., Din, G.: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In *Proceedings of the IFIP 17th Intern. Conf. on Testing Communicating Systems (TestCom 2005*), ISBN: 3-540-26054-4, Montreal, Canada. Springer-Verlag, 2005.

[ZMS07a] Zander-Nowicka, J., Marrero Pérez, A., Schieferdecker, I.: From Functional Requirements through Test Evaluation Design to Automatic Test Data Retrieval – a Concept for Testing of Software Dedicated for Hybrid Embedded Systems. In *Proceedings of the IEEE 2007 World Congress in Computer Science, Computer Engineering, & Applied Computing; SERP 2007*, Editors: Arabnia, H. R., Reza, H., Volume II, Pages: 347 – 353, ISBN: 1-60132-019-1, Las Vegas, NV, U.S.A. CSREA Press, 2007.

[ZMS07b] Zander-Nowicka, J., Marrero Pérez, A., Schieferdecker, I., Dai, Z. R.: Test Design Patterns for Embedded Systems. In *Business Process Engineering. Conquest-Tagungsband 2007 – Proceedings of the 10th International Conference on Quality Engineering in Software Technology*, Editors: Schieferdecker, I., Goericke, S., ISBN: 3898644898, Potsdam, Germany. dpunkt.Verlag GmbH, 2007.

[ZS07] Zimmermann, W., Schmidgall, R.: *Bussysteme in der Fahrzeugtechnik Protokolle und Standards*, ATZ-MTZ Fachbuch, ISBN: 9783834802354. Vieweg Friedr.+Sohn Verlag, 2007 (in German).

[ZSF06] Zander-Nowicka, J., Schieferdecker, I., Farkas, T.: Derivation of Executable Test Models From Embedded System Models using Model Driven Architecture Artefacts - Automotive Domain. In *Proceedings of the Model Based Engineering of Embedded Systems II (MBEES II)*, Editors: Giese, H., Rumpe, B., Schätz, B., TU Braunschweig Report TUBS-SSE 2006-01, Dagstuhl, Germany. 2006.

[ZSM06] Zander-Nowicka, J., Schieferdecker, I., Marrero Pérez, A.: Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems. In *Proceedings of the IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006)*, IEEE Catalog Number: 06CH37750C, ISBN: 1-4244-0052-X, ISSN: 1088-7725, Anaheim, CA, U.S.A. IEEE, 2006.

[ZVS$^+$07] Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 quality model to test specifications – exemplified for TTCN-3 test specifications. In *Proceedings Software Engineering 2007*, Editors: Bleek, W.-G., Raasch, J., Züllighoven, H., Pages: 231 – 244, ISBN: 978-3-88579-199-7. GI-LNI, 2007.

[ZXS08] Zander-Nowicka, J., Xiong, X., Schieferdecker, I.: Systematic Test Data Generation for Embedded Software. In *Proceedings of the IEEE 2008 World Congress in Computer Science, Computer Engineering, & Applied Computing; The 2008 International Conference on Software Engineering Research and Practice (SERP 2008)*, Editors: Arabnia H. R., Reza H., Volume I, Pages: 164 – 170, ISBN: 1-60132-086-8, Las Vegas, NV, U.S.A. CSREA Press, 2008.

Appendix A

List of Model-based Test Tools and Approaches

List of selected model-based test (MBT) approaches in the context of embedded systems relevant for the development of this thesis.

Table A: Classification of Selected Test Approaches Based on the MBT Taxonomy.

Test Approach, Tool	*Test Generation Selection Criteria and Technology*	*Test Execution Options*	*Test Evaluation Specification and Technology*	*Description*	*Link*
CTE/ES	- data coverage - requirements coverage - test case specification - manual generation - offline generation	- does not apply[46] - non-reactive	- does not apply here as the test evaluation is not supported at all	Classification Tree Editor for Embedded Systems (CTE/ES) implements the Classification Tree Method (CTM) [Con04a]. The SUT inputs form the classifications in the roots of the tree. Then, the input ranges are divided into classes according to the equivalence partitioning method. The test cases are specified by selecting leaves of the tree in the combination table. A line in the table specifies a test case. CTE/ES provides a way of finding test cases systematically. It breaks the test scenario design process down into steps. Additionally, the test scenario is visualized in a GUI.	www.razorcatdevelopment.de

[46] Unless otherwise noted, the expression *'does not apply'* is used when the test approach does not explicitly name the particular option or when the option does not matter in the context of a particular approach. In that case further deep investigation is needed to assess the option.

Embedded *Validator*	- does not apply - automatic generation - model checking - offline generation	- MiL, SiL - non-reactive	- requirements coverage - manual specification - does not apply	Embedded*Validator* [BBS04] is the model verification tool used for verifying temporal and causal safety-critical requirements of models designed in SL/SF and TargetLink. The method offers a set of test behavior patterns like *"an output is set only after certain input values are observed"* based on model checking. It is limited mainly to discrete model sectors. The actual test evaluation method offers a basic set of constraints for extracting discrete signal properties.	www.osc-es.de
JUMBL	- random and stochastic criteria - automatic generation - offline generation	- MiL, SiL[47] - non-reactive	- requirements coverage - automatic specification - offline evaluation	The Java Usage Model Builder Library (JUMBL) can generate test cases as a collection of test cases which cover the model with the minimum cost, by random sampling with replacement, by probability, or by interleaving the events of other test cases. The usage models are finite-state, time homogeneous Markov chains, characterized as deterministic finite automata with probabilistic transitions [Pro03]. There is also an interactive test case editor for creating test cases by hand. In [CLP08] the approach is used for testing SL/SF control models.	www.cs.utk.edu/sqrl/esp/jumbl.html
MaTeLo	- random and stochastic criteria - automatic generation - offline generation	- MiL, SiL - non-reactive	- requirements coverage - automatic specification - offline evaluation	Markov Test Logic (MaTeLo) tool can generate test suites according to several algorithms. Each of them optimizes the test effort according to the objectives such as boundary values, functional coverage, and reliability level. Test cases are generated in XML/HTML format for manual execution or in TTCN-3 [ETSI07] for automatic execution [DF03].	www.all4tec.net
MATT	- data coverage - automatic generation - offline generation	- MiL, SiL - non-reactive	- reference signals-based - manual specification - offline evaluation	MATLAB Automated Testing Tool (MATT) uses information that it obtains from ML/SL model in order to create a set of input test data. With a series of point and click selections the data can be set for each input port and parameters can be adjusted for accuracy, constant, minimum and maximum values. Once each input port has been set up, the test data matrix can be generated. The test matrix output is then returned to ML for simulation, code generation, comparison.	www.cs.umt.edu/RTSL/matt

[47] For SiL, PiL and HiL test adapters and test drivers are needed.

MEval	- does not apply since here back-to-back regression tests are considered.	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based - manual specification - offline evaluation	MEval offers an automatic comparison of test signals with their reference signals in ML/SL, provided that the reference signals are given. The tool applies innovative two-stage algorithms [WCF02]. Its strength is the successive use of the preprocessor and comparison component for signal evaluation.	www.itpower.de/meval.html
MiLEST	- data coverage - requirements coverage - test case specifications - automatic generation - offline generation	- MiL, extendable to SiL, PiL, HiL - reactive	- reference signal-feature – based - requirements coverage - test evaluation specifications - automatic and manual specification[48] - online evaluation	Model-in-the-Loop for Embedded System Test (MiLEST) is desirable for functional black-box testing of embedded hybrid software. A new method for the system stimulation and evaluation is supported, which breaks down requirements into characteristics of specific *signal features*. A novel understanding of *a signal* is defined that enables its description in an abstract way based on its properties (e.g., decrease, constant, maximum). Technically, MiLEST is a Simulink add-on built on top of the ML engine that represents an extension towards model-based testing activities. MiLEST consists of a library including callback functions, transformation functions, and other scripts. Reusable test patterns, generic validation functions and patterns for test data generators are provided. Transformations contribute to the automation of the test development process. Test data variants are created systematically and automatically. Reactive testing is also supported.	www.fokus.fraunhofer.de/en/motion/ueber_motion/index.html
MTest	- data coverage - requirements coverage - test case specification - manual generation - offline generation	- MiL, SiL, PiL, HiL - non-reactive	- reference signals-based - manual specification - offline evaluation	MTest [MTest] combines the classical module test with model-based development. The central element of the tool is the CTM and CTE/ES. It is integrated with SL and TargetLink. Once the test cases are designed in CTE/ES, MTest introduces such tasks as test development, test execution, test evaluation and test management. It enables SUT output signals to be compared with previously obtained reference signals using a reprocessing component and the difference matrix method. The reference signals can be defined using a signal editor or they can be obtained as a result of a simulation. MTest provides a means to automatically test automotive software within the whole development process. It is based on AutomationDesk's technology for test project management.	www.dspaceinc.com/ww/en/inc/home/products/sw/expsoft/mtest.cfm

[48] It depends on the process step when the evaluation design is developed (cf. Section 4.4 and Section 5.2 – 5.6).

PROVE-tech	- does not apply - manual generation - offline generation	- MiL, SiL, HiL - non-reactive	- does not apply - manual specification - offline evaluation	PROVEtech:Test Automation (PROVEtech:TA) realizes the test approach, which is not actually based on any model. However, it is a relevant tool in the context of automotive testing. It is an operational software, developed by the MBtech Group, for the control and automation of test systems, initially on HiL level. It constitutes a basis for executing automated tests on a real-time platform by means of its integrated development environment and program libraries for test execution on real-time computers.	www.mbtech-group.com/en
Reactis Tester	- structural model coverage - automatic generation - model checking *(Author decided to classify this approach as a sophisticated variant of model checking technology.)* - offline generation	- MiL, SiL, HiL - non-reactive	- test evaluation specifications - automatic specification - offline evaluation	Reactis Tester automatically generates test suites from SL/SF models. The idea behind this approach is to use guided simulation algorithms and heuristics so as to automatically obtain inputs covering the targets (i.e., model elements to be executed at least once). Two of the targets involve SL, three are specific to SF and the remaining include criteria within both the SL and the SF portions of a model. Each test case in a test suite consists of a sequence of inputs fed into the model as well as the responses to those inputs generated by the model. The obtained tests may be used for validating the model itself or for comparison of source-code implementation with model behavior results.	www.reactive-systems.com/tester.msp
Reactis Validator	- structural model coverage - requirements coverage - automatic generation - model checking - offline generation	- MiL, SiL - non-reactive	- test evaluation specifications - manual specification - online evaluation	Reactis Validator provides a test framework for validation of the system design. It enables to express the so-called assertions and user-defined targets graphically. The former check an SUT for potential errors. The latter monitor system behavior in order to detect the presence of certain desirable test cases [SD07]. If a failure occurs, a test execution sequence is delivered and it leads to the place where it happens. Then, this test is executed in Reactis Simulator.	www.reactive-systems.com/validator.msp

Safety Checker Blockset	- does not apply - automatic generation - model checking - offline generation	- MiL, SiL - non-reactive	- requirements coverage - manual specification - online evaluation	Safety-Checker Blockset (SCB) enables to formally verify properties of the SL/SF models, with the emphasis on SF. A property is a combination of model's variables connected to a proof operator. The verification mechanism is based on the model checking. Model checking analyzes a system regarding arbitrary input scenarios and can thus be viewed as a test performed against a formally specified requirement. SCB blocks typically express an unwanted situation (e.g., never together). Counter examples are provided in case of a failure.	www.tni-software.com/en/produits/safetytestbuilder/index.php
Safety Test Builder	- structural model coverage - requirements coverage - automatic generation - offline generation	- MiL, SiL - non-reactive	- does not apply - automatic specification - offline evaluation	Safety Test Builder (STB) is a solution dedicated to automating the production of test cases for embedded software, provided the software has been modeled using SL/SF. It generates test sequences covering a set of SL and SF test objectives based on the structural analysis. The approach creates lists of objectives automatically by model exploration, supporting basic coverage metrics. The test harness is created automatically. STB is dedicated to software testing at the function and subsystem level.	www.tni-software.com/en/produits/safetytestbuilder
SCADE Design Verifier	- automatic generation - model checking - offline generation	- MiL, SiL - non-reactive	- requirements coverage - manual specification - online evaluation	SCADE Design Verifier (DV) is a model-based proof engine allowing formal verification of safety-critical properties using model checking techniques. It enables proving of a design safety with respect to its requirement. In case of property verification failure, a counter example is provided. Safety properties are expressed using the SCADE language. A node implementing a property is called an observer. It receives the input variables involved in the property and produces an output that should be always true. DV is able to verify properties mixing boolean control logic, data-value transformations and temporal behavior. The core algorithms are based on Stalmarck's SAT-solving algorithm for dealing with boolean formulas, surrounded by induction schemes to deal with temporal behavior and state space search. These algorithms are coupled with constraint solving and decision procedures that handle the data path [DCB04].	www.esterel-technologies.com/products/scade-suite/design-verifier

Simulink® Validation and Verification™	- does not apply - manual generation	- MiL - non-reactive	- requirements coverage - manual specification - online evaluation	Simulink Validation and Verification (SL VV) is a tool for validating SL models. Tests are produced manually as a set of signals and can then be subjected to automated coverage analysis on the level of the model. The assertion blocks are set up to notify the user if a failure arises. A selection list of available assertions can be displayed in the SL Signal Builder, in order to activate or deactivate certain assertions depending on the input signals to be generated. SL VV enables traceability from requirements to SL/SF models and model coverage analysis. For SF charts, the classic state coverage and transition coverage are provided. SL blocks rely on dedicated criteria such as lookup table coverage, which records the frequency of table lookups in a block. Other structural coverage analysis is provided, i.e., for data coverage boundary values, signal range analysis and for complex boolean decisions (decision coverage, condition coverage, modified condition/decision coverage). Test cases are run on the model itself. To run tests on the SUT, the tests must be recorded first and then adapted to the SUT interface.	www.mathworks.com/products/simverification/
Simulink® Design Verifier™	- structural model coverage - automatic generation - theorem proving - offline generation	- MiL, SiL - non-reactive	- requirements coverage - test evaluation specifications - manual specification - online evaluation	Simulink Design Verifier generates tests for SL models that satisfy model coverage and user-defined objectives. After completing the generation it produces a test harness model that contains test cases. The tool also proves model properties and generates examples of their violations. It uses mathematical procedures to search through the possible execution paths of the model so as to find test cases and counter examples. The main blocks for specifying the objectives are proof assumption, proof objective, test condition, test objective and verification subsystem. A similar approach is realized in a research prototype, called automatic test-case generation (ATG) [GMS07].	www.mathworks.com/products/sldesignverifier

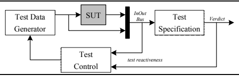
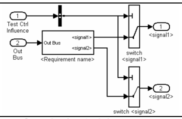
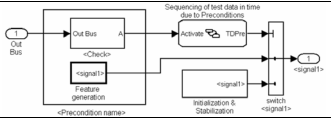
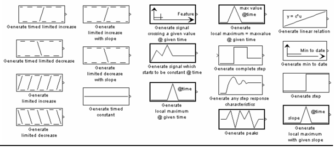
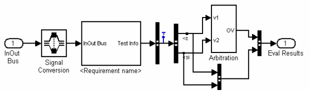
System Test™	- data coverage - automatic and manual generation - offline generation	- MiL, SiL, HiL - non-reactive	- reference signals-based - manual specification - offline evaluation	SystemTest is a tool for testing ML scripts and SL models. Test cases can be specified at a low abstraction level and are executed manually. The test vectors may be defined manually using ML expressions or generated randomly applying probability distributions for Monte Carlo simulation [DFG01]. Besides a small set of automatic evaluation means based on the reference signals, the actual test assessment is performed manually using graphical signal representations and tables. SystemTest supports test case reusability. Parameter sweep for system optimization is also possible.	www.mathworks.com/products/systemtest
Testing of Auto-Focus Models [Pre03b]	- data coverage - test case specifications - automatic generation - symbolic execution - offline generation	- does not apply - non-reactive	- test evaluation specifications - automatic specifications - does not apply	The approach enables both generation of functional, but also structural (based on MC/DC criterion) test specification [Pre03, Pre03b], followed by concrete test cases derivation. It is based on AutoFocus system models. The generation of test is supported by the symbolic execution on the grounds of CLP (initially transformed from the AutoFocus models). [Pre04] concludes that test case generation for both functional and structural test case specifications boils down to finding states in the model's state space. The aim of a symbolic execution of a model is then to find a trace – a test case – that leads to the specified state. Specification of test case in the form of interaction patterns means providing the concrete signals. For universal properties such as invariants, the deduction of test case specifications is possible by syntactically transforming temporal logic formulas [Pre03b].	tumb1.biblio.tu-muenchen.de/publ/ diss/in/2003/pretschner.pdf
TPT	- data coverage - requirements coverage - test case specification - manual generation - offline and online generation	- MiL, SiL, PiL, HiL - reactive	- reference signal-feature – based - manual specification - online and offline evaluation	The objectives of Time Partitioning Testing (TPT) are to support a test modeling technique that allows the systematic selection of test cases, to facilitate a precise, formal, portable, but simple representation of test cases for model-based automotive developments, and thereby, to provide an infrastructure for automated test execution and automated test assessments even for real-time environments [LK08]. TPT supports the selection of test data on the semantic basis of so-called testlets and several syntactic techniques. Testlets facilitate an exact description of test data and guarantee the automation of test execution and test evaluation. Test evaluation is based on the concept of the property of a signal. A library containing several evaluation functions is available. External tools can be easily integrated into the evaluation process too.	www.piketec.com/products/tpt.php

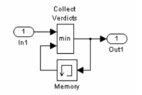
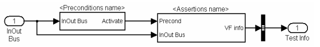
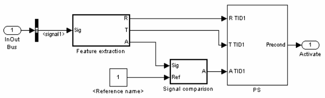
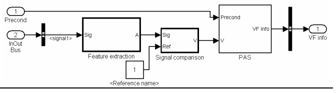
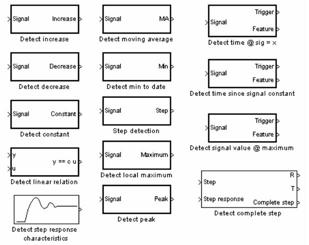
T-VEC Tester for Simulink	- structural model coverage - data coverage - requirements specification - automatic generation - offline generation	- MiL, SiL - non-reactive	- test case specifications [ROT98] - automatic specification - does not apply	Test VECtor (T-VEC) Tester automates some steps of the test development process by analyzing the structure of the SL model. Test cases for validating the model and testing implementations of the model are determined. The test selection process produces the set of test vectors effective in revealing both decision and computational errors in logical, integer and floating-point domains [BBN04]. T-VEC determines test inputs, expected outputs and a mapping of each test to the associated requirement, directly from SL specifications. T-VEC analyzes also the transformed specification to determine whether all specification elements have a corresponding test vector.	www.t-vec.com/solutions/simulink.php

Appendix B

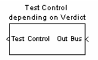
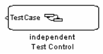
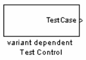
Test Patterns Applicable for Building the Test System

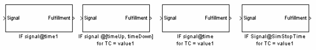
Table B: Test Patterns Implemented in MiLEST.

Test Activity* / *Test System Abstraction Level	***Test Pattern Name***	***Context***	***Problem***	***Solution Instances***
		Test Harness Preparation		
Test Harness Level	Test harness	Functional test	Generation of a test frame around the SUT.	 Test Data Generator SUT InOut Bus Test Specification Verdict Test Control test reactiveness
		Test Data Generation		
Test Requirement Level	Collection of the test requirements	Instantiation of a test requirement	Generation of a frame for collecting the test requirements	 1 Test Ctrl Influence 2 Out Bus Out Bus <signal1> <signal2> <Requirement name> switch <signal1> 1 <signal1> 2 <signal2> switch <signal2>
Test Case Level	Collection of the test cases	Specification and sequencing of stimuli for a test case	Generation of a block sequencing the test stimuli along the test cases	 Sequencing of test data in time due to Preconditions 1 Out Bus Out Bus A <Check> <signal1> Feature generation <Precondition name> Activate TDPre <signal1> Initialization & Stabilization switch <signal1> 1 <signal1>
Feature Generation Level	Generate SigF	Generation of concrete signals along the test cases	Generation of the SigF to stimulate the SUT	 Generate timed limited increase Generate limited increase with slope Feature Generate signal crossing a given value @ given time max value @time Generate local maximum = maxvalue @ given time y = c*u Generate linear relation Generate timed limited decrease Generate limited decrease with slope Generate signal which starts to be constant @ time Generate complete step Min to date Generate min to date Generate limited increase Generate timed constant @time Generate local maximum @ given time Generate any step response characteristics Generate step Generate limited decrease Generate peaks slope @time Generate local maximum with given slope
		Test Specification and Test Evaluation		
Test Requirement Level	Collection of the test requirements	Technical instantiation of a test requirement	Tracing a set of test requirements in the form of their abstract instantiation	 1 InOut Bus Signal Conversion InOut Bus Test Info <Requirement name> <s <si v1 v2 OV Arbitration 1 Eval Results

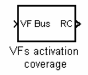
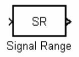
Test Requirement Level	Arbitration	Delivering the test results	Extraction of an *overall verdict* from the collection of *local verdicts*	
Validation Function Level	A validation function block	Specification of an abstract test scenario	Decomposition of an abstract test scenario into a set of preconditions and assertions	 <Preconditions name> <Assertions name> InOut Bus Activate Precond VF info Test Info
Validation Function Level	Collection of preconditions	Specification of the preconditions	Decomposition of the preconditions into a set of SigFs to be extracted	 InOut Bus <signal1> Sig R T A Feature extraction <Reference name> Ref Signal comparison R TID1 T TID1 A TID1 Precond PS Activate
Validation Function Level	Collection of assertions	Specification of the assertions	Decomposition of the assertions into a set of SigFs to be extracted	 Precond InOut Bus <signal1> Sig A Feature extraction Ref V Signal comparison <Reference name> VF Info PAS VF info
Feature Detection Level	Detect SigF characteristics	Evaluation of a mathematical function	Assessment of a control unit behavior in terms of a selected SigF	 Signal Increase Detect increase MA Detect moving average Trigger Feature Detect time @ sig = x Decrease Detect decrease Min Detect min to date Detect time since signal constant Constant Detect constant Step Step detection Detect signal value @ maximum y u y == c u Detect linear relation Maximum Detect local maximum Step response R T Complete step Detect complete step Peak Detect peak Detect step response characteristics

Test Control

Test Harness Level	Test control depending on verdict value	Specification of the test control	Specification of such a test control where the sequencing of test cases depends on the selected verdict values	 Test Control depending on Verdict Test Control Out Bus
Test Harness Level	Independent test control	Specification of the test control	Specification of such a test control where no dependencies between test cases exist	
Test Harness Level	Variants dependent test control	Specification of the test control	Specification of such a test control where the sequencing of test cases depends on the number or value of test data variants applied in a selected test case(s)	

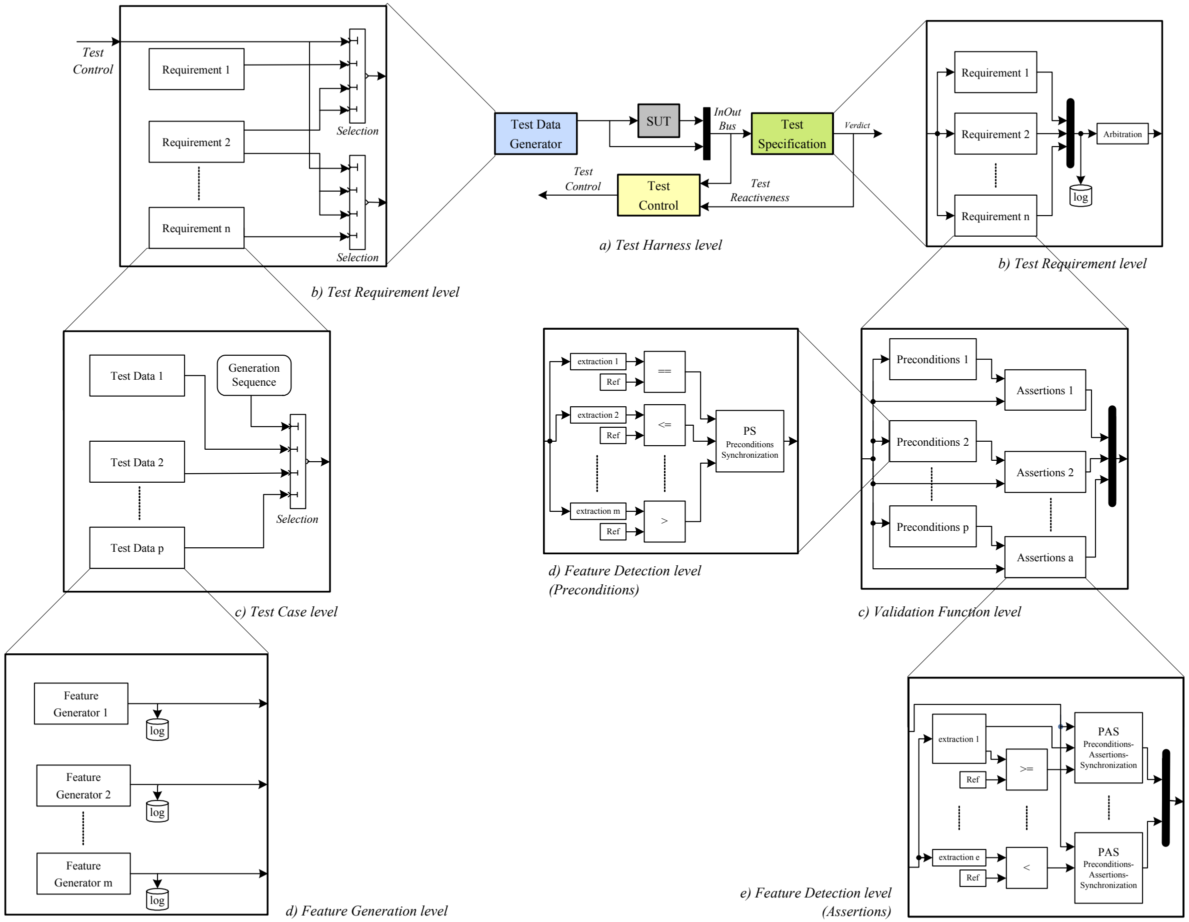
Test Control Level	Test control condition	Specification of the test control conditions	Specification of default conditions enabling to constrain the definition of a test control	 Signal
				Fulfillment
				IF signal@time1
				IF signal @[timeUp, timeDown] for TC = value1
				IF signal@time for TC = value1
				IF Signal@SimStopTime for TC = value1

Test Quality Assessment

Test Harness Level	VFs activation coverage	Assessment of the quality of the test specification	Evaluation of the test specification effectivity and efficiency by checking the activation coverage of validation functions	
Test Harness Level	Signal range	Assessment of the quality of the test specification	Evaluation of the SUT input/output signal range coverage	

Appendix C

Hierarchical Architecture of the Test System



a) Test Harness level

b) Test Requirement level

b) Test Requirement level

c) Test Case level

d) Feature Detection level (Preconditions)

c) Validation Function level

d) Feature Generation level

e) Feature Detection level (Assertions)

Appendix D

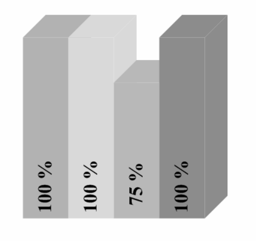
Questionnaire

Imagine that you have been a system and test engineer in an international software engineering company for the last five years. Now, you got promoted and you are becoming a leader of the test engineers' group in an automotive company. The task assigned to your team is to test the functionality of the electronic control units in a car and you are responsible for the success of the project.

Additionally, assume that the Simulink®/Stateflow® (SL/SF) modeling language is applied for building the system under test.

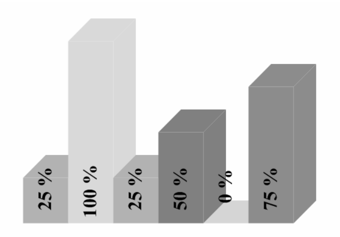
1. Which of the following black-box test approaches do you know?

□ State charts for testing, e.g., Time Partitioning Testing
□ Classification Tree Method (CTM) - based testing, e.g., CTM for Embedded Systems
□ SL/SF add-in for testing, e.g., SL Design Verifier, MiLEST
□ Sequence diagrams for testing, e.g., using UML® Testing Profile for Embedded Systems



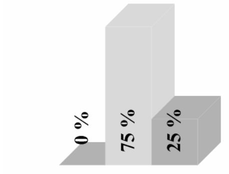
2. Which modeling technique would you prefer to apply?

□ State charts for testing
□ CTM - based testing
□ SL/SF add-in for testing
□ Sequence diagrams for testing
□ Your own method. Which?
□ More of them. Why?



3. How much time would you expect to need for getting familiar with the method?

□ a few days
□ a few weeks
□ a few months



Appendix E

Contents of the Implementation

Test system library:

`Path:/MiLEST library`

`MiLEST.mdl`	MiLEST library

Functions for test specification:

`Path:/Transformation`

`PAS callback.m`	PAS callback function.
`PAS init.m`	PAS initialization function.
`PS init.m`	PS initialization function.
`PSTDD callback.m`	PS TDD callback function.
`PSTI callback.m`	PS TI callback function.
`PSTID callback.m`	PS TID callback function.
`slblocks.m`	Definition of the Simulink library block representation.
`DelDelay.m`	Elimination of verdict delays.

Transformation functions for test data derivation:

`Path:/Transformation`

`GenConstVar_Single_D_1.m`	Partitioning of *Generate Decrease* produced for extraction of *Decrease*.
`GenDecrVar_Single.m`	Partitioning of *Generate Decrease* produced for extraction of *signal<=x*.
`GenDecrVar_Single_D.m`	Partitioning of *Generate Constant* produced for extraction of *Constant*.
`GenIncrVar_Single.m`	Partitioning of *Generate Increase* produced for extraction of *signal>=x*.
`GenIncrVar_Single_D.m`	Partitioning of *Generate Increase* produced for extraction of *Increase*.
`GenVarSequence_1.m`	Synchronization of the variants using Stateflow diagram.
`TestcontrolGen.m`	Synchronization of the test cases execution in the TDG using *one factor at a time* combination strategy.
`TestcontrolGen_v.m`	Synchronization of the test cases execution in the TDG using *minimal* combination strategy.
`Testdata_Preconditions_G.m`	Transformation at the *Test Case Level* – TDGen View.
`TestdataFeature_G.m`	Transformation at the *Feature Generation Level* – TDGen View.
`TestdataGen.m`	Transformation at the *Test Harness Level* – TDGen View.
`TestdataReqG.m`	Transformation at the *Test Requirement Level* – TDGen View.

Other functions:

`Path:/Transformation`

`TransformationStep4.m`	Transformation of pure SUT model to test harness.
`system_name2.m`	Callback function for the mask parameter: *number of requirements* of the <*system name*> system.
`shut_mask.m`	Shutting the mask off in the <*Test data generator*> system.
`Test_D_Gen.m`	Callback function for the mask parameter: *number of requirements* of the <*Test data generator*> system.
`Test_D_Gen_S.m`	Callback function for the mask parameter: *number of signals* of the <*Test data generator*> system.
`VFs_callback.m`	Callback function for the mask parameter of the <*Requirement name*> system – TSpec View.
`GenLogdata_1.m`	Setting the parameter of *'DataLogging*’ on ‘*on*’ in all preconditions. This function assists the calculation of input coverage.
`GenLogdata_tc.m`	Setting the parameter of *'DataLogging*’ on ‘*on*’ in all requirements. This function assists the calculation of input coverage.
`get_partition.m`	Calculating the partition coverage.
`input_coverage_1.m`	Listing of partition coverage for all SUT input signals.
`output_coverage_1.m`	Listing of partition coverage for all SUT output signals.
`ReqName_callback.m`	Callback function for the mask parameter of the <*Requirement name*> system.
`fix_pos.m`	Graphical adjustment of the position between two blocks in two dimensions.
`Mask_Shut_off.m`	Shutting the mask off in the <*Test data generator*> system, after transformation.

Examples:

`Path:/Adaptive Cruise Control`

`pedal.mat`	Pedal characteristic.
`drossel.mat`	Throttle characteristic.
`fzgbib.mdl`	Vehicle model library.
`tempomat_para.m`	Speed Control test parameters.
`ACC.mdl`	Adaptive Cruise Control model.
`ACC_FM.mdl`	Adaptive Cruise Control model including the failure management.
`ACC_Test.mdl`	Adaptive Cruise Control model including the entire test system.

`Path:/Speed Controller`

`Speed_Controller_Test.mdl`	Speed Control model including the test specification part.

`Path:/Pedal Interpretation`

`Pedal_Interpretation_TC.mdl`	Pedal Interpretation model including the test specification part.
`Pedal_Interpretation_Test.mdl`	Pedal Interpretation after the transformation including the entire test system.
`TestReporter.pdf`	Test report document.
`TestReporter.rpt`	Test report generator.

Application of I/O Parameters for the Transformations Functions:

.m File ID	*I/O Parameters*	*Example*
fix_pos	I: blocka, blockb, x, y	fix_pos('Pedal_Interpretation_Test/Bus Selector', 'Pedal_Interpretation_Test/Memory',120,80);
GenConstVar_Single_D_1	I: is_increase, tar, InputName, ODT, duration_tick O: variants_nr (i.e., *number of variants*)	variants_nr = GenConstVar_Single_D_1(['Pedal_Interpretation_Test/ Generate constant ' num2str(1)], 'Pedal_Interpretation_Test', 'phi_Acc', 'OutDataTypeMode', '200');
GenDecrVar_Single_D	I:is_increase, tar, InputName, duration_tick O: variants_nr	variants_nr = GenDecrVar_Single_D ('Pedal_Interpretation_Test/Generate constant ', 'Pedal_Interpretation_Test' , 'phi_Acc', '200');
GenDecrVar_Single	I: is_decrease, tar, InputName, ref, duration_tick O: variants_nr	variants_nr = GenDecrVar_Single ('Pedal_Interpretation_Test/Generate constant ', 'Pedal_Interpretation_Test' , 'phi_Acc', '5', '200');
GenIncrVar_Single	I:is_increase, tar, InputName, ref, duration_tick O: variants_nr	variants_nr = GenIncrVar_Single ('Pedal_Interpretation_Test/Generate constant ', 'Pedal_Interpretation_Test' , 'phi_Acc', '5', '200');
GenIncrVar_Single_D	I:is_increase, tar, InputName, duration_tick O: variants_nr	variants_nr = GenIncrVar_Single_D ('Pedal_Interpretation_Test/Generate constant ', 'Pedal_Interpretation_Test' , 'phi_Acc', '200');
GenLogdata	I: targetModel	GenLogdata('Pedal_Interpretation_Test')
GenLogdata_tc	I: targetModel	GenLogdata_tc('Pedal_Interpretation_Test')
GenVarSequence_1	I: path, ReqNr	GenVarSequence_1 ('Pedal_Interpretation_Test/variants sequence', 4)
get_partition	I: lower, upper, partitionPoint, actualSignalRange O: par (i.e., *actual number of partitions*), partition (i.e., *expected number of partitions*)	[p partition] = get_partition(-10, 70, [0], [-10 70]);
input_coverage_1	I: targetModel	input_coverage_1('Pedal_Interpretation_Test')
output_coverage_1	I: targetModel	output_coverage_1('Pedal_Interpretation_Test')
ReqName_callback	I: blockName	ReqName_callback(gcb)
TestcontrolGen_v	I: targetModel, time	TestcontrolGen_v('Pedal_Interpretation_Test', 200);
TestcontrolGen	I: targetModel	TestcontrolGen_v('Pedal_Interpretation_Test', 400);
Testdata_Preconditions_G	I: sys, RequirementName, tar, duration_tick O: TCD (i.e., *test case duration*), p_nr (i.e., *number of preconditions*), vr_max (i.e., *maximum number of variants in the requirement*)	[t1 p_nr vr_max_1] = Testdata_Preconditions_G('Pedal_Interpretation_Test/ Validation Functions', 'SRPI01.1', 'Pedal_Interpretation_Test_TC', '200');
TestdataFeature_G	I: VFsName, TestDataName, tar, duration_tick O: vr_max	vr_max_1 = TestdataFeature_G('Pedal_Interpretation_Test/Validation Functions/SRPI01.1/v=const', 'Pedal_Interpretation_Test_TC/TestData1/SRPI01.1/v=const','Pedal_Interpretation_Test_TC', '200');
TestdataGen	I: source, SUT, duration_tick O: time, vr_max	[time vr_max] = TestdataGen('Pedal_Interpretation_Test', 'PedalInterpretation','200')
TestdataReqG	I: sys, block, tar, duration_tick O: time_exe, vr_max	[time vr_max_1] = TestdataReqG(' Pedal_Interpretation_Test/Validation Functions','Test Info', 'Pedal_Interpretation_Test_TC', '200');

Appendix F

CURRICULUM VITAE – JUSTYNA ZANDER-NOWICKA

PERSONAL DATA:

BIRTHDAY/-PLACE:	February 28th, 1980, Elbląg (Poland)

EDUCATION:

SINCE 2004	Junior and Senior Researcher at the Fraunhofer Institute FOKUS, Berlin (Germany)
OCT. –NOV. 2008	Visiting Researcher at The MathWorks™, Inc., Natick, MA (U.S.A.)
SEPT.–OCT. 2007	Visiting Scholar at the University of California in San Diego (U.S.A.), Computer Science and Engineering Department
2004–2005	Studienkolleg zu Berlin, Interdisciplinary European Studies (Germany)
JULY–SEPT. 2003	Student Researcher at the Fraunhofer Institute FIRST, Berlin (Germany)
2003–2005	Technical University Berlin (Germany), Faculty IV – Electrical Engineering and Computer Science, Department for Design and Testing of Telecommunications Systems; MASTER OF SCIENCE
2001–2004	University of Applied Sciences in Elbląg (Poland), Institute of Applied Computer Science, Databases and Software Engineering; BACHELOR OF SCIENCE
1999–2003	Gdańsk University of Technology (Poland), Chemistry Faculty, Department of Environmental Protection and Management, Chemical Systems of Environmental Protection; BACHELOR OF SCIENCE

Index

A

abstraction level 7, 25, 97, 98, 197
  feature detection 106
  feature generation 104, 160
  test case 102, 147, 160, 169
  test harness 99, 157
  test requirement 99, 158
  validation function 102, 146, 159, 167
actuator 11
adaptive cruise control 141
alternative 114
analysis
  boundary value 35
arbitration 92, 99, 101, 112, 114, 197
ASCET 17
assertions set 92
AutoFocus 37
automotive 28, 40
AUTOSAR 41, 195

B

back-to-back test 45
basic signal feature
  constant 66
  continuity of the derivative 66
  decrease 66
  increase 66
  inflection point 70
  linear functional relation 68
  local maximum 70
  local minimum 70
  maximum to date 68
  minimum to date 68
  signal value 65
boundary testing 120

C

causal system 54
Charon 17
classification
  of model-based testing 32
  of signal features 62
  of test approaches 47
classification tree method 35
closed-loop system 26
code generation 18
combination strategy 124
  minimal combination 124
  n-wise combination 125
  one factor at a time 125
  pair-wise combination 125
component in-the-loop testing 156
consistency
  of a test 177
continuous system 17
control theory 14
correctness
  of a test 177
coverage
  of data 35
  of error detection 124
  of model 35, 182
  of requirements 35, 39
  of test 178
CTE/ES 35, 40

D

Daimler AG 41
data coverage 35
data type boundary 120
development
  service-oriented 133
development process
  model-based 16
Dymola 17
dynamic testing 23, 90

E

electronic control unit (ECU) 13
embedded software 12
embedded system 12

Embedded*Validator* 37, 45
equivalence class 35
error
  detection coverage 124
execution platform
  HIL 25
  MIL 25
  PIL 25
  SIL 25

F

feature signal 60
feedback 14
Finite State Machine (FSM) 19
fourth order Runge-Kutta formula 20
functional model 17
functional testing 24

G

generation rule 121
global clock 133

H

High level *hySCt (HhySCt)* 164
HiL 25
hybrid Sequence Charts 133
hybrid Sequence Charts for testing (hySCt) 164

I

IEC 61508 22
IF – THEN rule
  for test data generation 93
  for test specification 92, 105, 112, 114, 146, 180, 199
implementation model 17
increase generation 122
integration level testing 27, 124, 133, 164
ISO 26262 22

L

library
  in Simulink 18
logical connective 56

M

MATLAB 18
MATLAB Automated Testing Tool 35
MATLAB/Simulink/Stateflow 3, 11, 17, 18
mean value testing 120
Message Sequence Charts (MSC) 133
MEval 45
MiL 25
MiLEST limitations 188
minimal combination 124
model
  functional 17
  implementation 17
  of a system 5, 30
model checking 37
model coverage 35, 182
Model Coverage Tool 182
Model Driven Architecture 36, 137
model-based 3, 16
  development 3, 16
  testing (MBT) 29
model-driven testing (MDT) 29
Modelica 17
Model-in-the-Loop for Embedded System Test (MiLEST) 7, 41, 88, 114, 191
modus tollens rule 113
monitoring 14, 26
MTest 137
multiple V-model 16

N

non-functional testing 24
n-wise combination 125

O

one factor at a time 125
open-loop system 26

P

pair-wise combination 125
paradigm
  of modeling 33
  of testing based on signal feature 89
paradigm shift 3
partition point 120
partitioning
  of SUT input 123
  of SUT output 123
pedal interpretation 144
PiL 25
preconditions set 92, 197
process
  correct-by-construction 18
  external 14
  in MiLEST 90
  of test 22
proportional-integral-derivative (PID) controller 15

Q

quality assurance 133
Query/View/Transformation (QVT) 137

R

random testing 120
rapid control prototyping 19
Reactis Tester 35
Reactis Validator 46
reactiveness of a test 26, 129
redundancy 135
reference signal 38
representative 121
requirements coverage 35, 39
reset signal 61
role 134
rules
  of generation 121
  of transformations 118
Runge-Kutta formula 20

S

Safety Test Builder 35
sample time 54
SCADE 17
sensor 11
separation of concerns 17, 54, 116
sequencing of variants 126
service 133
S-function 18
short time Fourier transform 70
signal 53
  continuous 53
  discrete 53
  evaluation 54, 59
  feature 60
  generation 54, 59
  processing 54
  reset 61
  trigger 60
signal range 120
signal feature 55, 197
  classification 62
  conversion 106
  identifiable with delay 63, 70
  identifiable without delay 63, 70
  non-triggered 63
  triggered 70
  triggered identifiable with indeterminate delay 77
SiL 25
simulation in Simulink 20
simulation time step size 54
Simulink 18
Simulink Design Verifier 37, 46
Simulink library 18, 117
Simulink simulation 20
Simulink Verification and Validation 39, 182
software-intensive system 11
solver 20
  fixed-step 20
  ode4 20
  variable-step 20
speed controller 156
Stateflow 19
static testing 23
step response 80
structural testing 24
symbolic execution 37
synchronization 106
  of preconditions 92, 106, 197
  of preconditions and assertions 92, 106
  of test stimuli 91
system
  causal 54
  continuous 17
  embedded 11
  hybrid 12
  reactive 12
  real-time 13
  software-intensive 11
  under test 11
system configuration
  closed-loop 26
  open-loop 26
system design 5, 30
system model 5, 30, 198
systematic testing 28, 48, 88, 90, 92, 98, 120, 137
SystemTest 38, 45

T

taxonomy
  of model-based testing 32
temporal expression 57
test approach 43
test assessment 38, 92, 198
test case 124, 198
test data selection criteria
  boundary values 120
  equivalence classes 120
  mean values 120
  random 120
test dimensions 23
test evaluation 38, 92, 198
test execution 37, 136
test generation 34
test harness 99, 198
test implementation 22
test modeling guidelines 182

test oracle 38
test pattern 94
  test control 91, 132
  test data generation 91, 102
  test harness 91, 99
  test specification 91, 102
  validation function 102
test quality metric 178
  test control related 181
  test data related 178
  test specification related 180
test quality model 178
test reactiveness 26, 129
test selection criteria 34
test specification 22, 112
  assertions set 92
  preconditions set 92, 197
  validation function 92, 146, 147, 159, 168
test step 124
test strategy 187
test suite 124
testing
  back-to-back 45
  boundary 120
  dynamic 23, 90
  functional 24
  mean value 120
  model-based 29
  model-driven 29
  non-functional 24
  of component 27
  of component in-the-loop 27, 156
  of software 21
  of system 27
  on integration level 27, 124, 133, 164
  random 120
  reactive 26
  static 23
  structural 24
  systematic 28, 48, 88, 90, 92, 98, 120, 137
TestVECtor 45
theorem proving 37
Time Partitioning Testing (TPT) 137
tools
  ASCET 17
  ATG 43
  AutoFocus 37
  CTE/ES 35, 40
  Embedded*Validator* 37, 45
  LabView 17
  MATLAB Automated Testing Tool 35
  MEval 45
  Model Coverage Tool 182
  MTest 40
  Reactis Tester 35
  Reactis Validator 46
  Reactive Test Bench 38
  Report Generator 136
  Safety Checker Blockset 37
  Safety Test Builder 35
  SCADE 17
  Simulink Design Verifier 37, 46
  Simulink Verification and Validation 39, 182
  SystemTest 38, 45
  Testing-UPPAAL 42
  TestVECtor 45
  TPT 40
traceability 29, 35, 98, 104
transformation
  of IF-THEN rule 193
transformation rules 118
transposition rule 113
trigger signal 60
TTCN-3
  continuous 41
  core language 36, 41, 194
  embedded 41, 95

U

UML Testing Profile 35
  for Embedded Systems 41, 137
Unified Modeling Language (UML) 17

V

validation 22
validation function 92, 146, 147, 159, 168
variants sequencing 126
verdict 112
verification 22
V-Modell 16

W

watchdog 28, 40, 46