MODEL-CHECKING AND MODEL-BASED TESTING OF AUTOMOTIVE EMBEDDED SYSTEMS
STARTING FROM THE SYSTEM ARCHITECTURE

Raluc Marinescu
2014
Abstract

Nowadays, modern vehicles are equipped with electrical and electronic systems that implement highly complex functions such as anti-lock braking or cruise control. The use of such embedded systems in the automotive domain requires a development process that takes into account their complex features. In this context, architectural models have been introduced in system development as convenient abstractions of the system's structure represented by interacting components. To enjoy the full benefits of such abstractions, the architectural models should be complemented by analysis frameworks that provide means for formal verification, and ideally also model-based testing, tailored to complex automotive systems. One major difficulty in developing such frameworks lies in the fact that architectural models represent the system's structure as well as inter-component communication, often without the actual description of the behavior. This entails the need to integrate the two views (structural and behavioral) in order to integrate them in a formal framework for verification.

In this thesis, we propose an integrated formal modeling and analysis methodology for automotive embedded systems that are originally described in the domain-specific architectural language E AST-ADL. Our analysis methodology relies on formal verification of the original E AST-ADL model by model-checking with U PPAAL PORT for component-based analysis, and U PPAAL SMC for statistical model-checking. To enable analysis, we first propose a formal description of the E AST-ADL components as networks of timed automata, which are U PPAAL's modeling language. Since code implementation is in fact what is deployed on the vehicle, it is highly desirable to narrow the gap between the code and the architectural model, but also to test the implementation for various requirements. To accomplish the former, we define an executable semantics of the UPPAAL PORT components. To be able to support testing of E AST-ADL based implementations, we take advantage of the model-checker's ability to generate witness traces during verification of reachability properties. Consequently, we employ UPPAAL PORT to generate such traces that become our abstract test-cases. By pairing the automated model-based test-case generator with an automatic transformation from the abstract test-cases to Python scripts, we enable the execution of the generated Python
Abstract

Nowadays, modern vehicles are equipped with electrical and electronic systems that implement highly complex functions such as anti-lock braking or cruise control. The use of such embedded systems in the automotive domain requires a development process that takes into account their complex features. In this context, architectural models have been introduced in system development as convenient abstractions of the system’s structure represented by interacting components. To enjoy the full benefits of such abstractions, the architectural models should be complemented by analysis frameworks that provide means for formal verification, and ideally also model-based testing, tailored to complex automotive systems. One major difficulty in developing such frameworks lies in the fact that architectural models represent the system’s structure as well as inter-component communication, often without the actual description of the behavior. This entails the need to integrate the two views (structural and behavioral) in order to integrate them in a formal framework for verification.

In this thesis, we propose an integrated formal modeling and analysis methodology for automotive embedded systems that are originally described in the domain-specific architectural language EAST-ADL. Our analysis methodology relies on formal verification of the original EAST-ADL model by model-checking with UPPAAL PORT for component-based analysis, and UPPAAL SMC for statistical model-checking. To enable analysis, we first propose a formal description of the EAST-ADL components as networks of timed automata, which are UPPAAL’s modeling language. Since code implementation is in fact what is deployed on the vehicle, it is highly desirable to narrow the gap between the code and the architectural model, but also to test the implementation for various requirements. To accomplish the former, we define an executable semantics of the UPPAAL PORT components. To be able to support testing of EAST-ADL based implementations, we take advantage of the model-checker’s ability to generate witness traces during verification of reachability properties. Consequently, we employ UPPAAL PORT to generate such traces that become our abstract test-cases. By pairing the automated model-based test-case generator with an automatic transformation from the abstract test-cases to Python scripts, we enable the execution of the generated Python
scripts (our concrete test-cases) on the system under test. The entire formal analysis and model-based testing framework is one solution to analyzing EAST-ADL models by model-checking techniques. We show the framework’s applicability on an automotive industrial prototype, namely a Brake-by-Wire system.
“Inspiration is for amateurs -
the rest of us just show up and get to work.”
Chuck Close
Acknowledgments

First and foremost, I would like to thank my two amazing supervisors, Cristina Seceleanu and Paul Pettersson. Your passion for research, your patience, your positive and energetic attitude, is truly inspiring. Thank you for giving me the opportunity to become a PhD student, and thank you for your guidance and support. You are the best role models a girl can have!

Many thanks to my family for their love and support through my long decade of university studies. You've taught me that studying is important, and you have helped me to follow my dreams, even though that meant to live so far apart from each others. Also, I would like to thank my future husband, Eduard, for believing in me and being there for me through the hardest times. Your love for testing and your contagious optimism make our discussions about work a delight!

I am grateful to all the wonderful researchers at Mälardalen University, both senior researchers and fellow PhD students, for all the wonderful moments we have shared (and we will share) together during lectures, research meetings, and coffee brakes.

Finally, I'd like to thank VINNOV A and ARTEMIS JU whose financial support via the MBAT research project (grant agreement number 269335) has made this thesis possible.

Raluca Marinescu
Västerås, Sweden
November 21, 2014
Acknowledgments

First and foremost, I would like to thank my two amazing supervisors, Cristina Seceleanu and Paul Pettersson. Your passion for research, your patience, your positive and energetic attitude, is truly inspiring. Thank you for giving me the opportunity to become a PhD student, and thank you for your guidance and support. You are the best role models a girl can have!

Many thanks to my family for their love and support through my long decade of university studies. You’ve taught me that studying is important, and you have helped me to follow my dreams, even though that meant to live so far apart from each others. Also, I would like to thank my future husband, Eduard, for believing in me and being there for me through the hardest times. Your love for testing and your contagious optimism make our discussions about work a delight!

I am grateful to all the wonderful researchers at Mälardalen University, both senior researchers and fellow PhD students, for all the wonderful moments we have shared (and we will share) together during lectures, research meetings, and coffee brakes.

Finally, I’d like to thank VINNOVA and ARTEMIS JU whose financial support via the MBAT research project (grant agreement number 269335) has made this thesis possible.

Raluca Marinescu
Västerås, Sweden
November 21, 2014
List of Publications

Papers Included in the Licentiate Thesis


1The included papers have been reformatted to comply with the thesis layout.
List of Publications

Papers Included in the Licentiate Thesis


1The included papers have been reformatted to comply with the thesis layout.
Other Publications


# Contents

I  Thesis  
1  Introduction  
   1.1  Thesis overview  
2  Preliminaries  
   2.1  Model-driven Development of Systems  
   2.2  Architectural Modeling of Automotive Systems: EAST-ADL  
   2.3  Formal Analysis by Model-checking  
      2.3.1  Timed Automata Models  
      2.3.2  Model-checking of Timed Automata Descriptions  
   2.4  Model-based Testing  
3  Research Method  
4  Research Goals  
   4.1  Problem Description  
   4.2  Research Goals  
5  Thesis Contribution  
   5.1  Formal Verification of EAST-ADL Models  
      5.1.1  Formal Semantics of EAST-ADL as Timed Automata  
      5.1.2  Model-checking EAST-ADL as Timed Automata Networks  
   5.2  Tools for Model-based Testing: an Overview  
   5.3  Implementation guidelines for EAST-ADL+TA Models  
   5.4  Model-based Testing Starting from EAST-ADL  
      5.4.1  Generation of Abstract Test-Cases  
      5.4.2  Generation and Execution of Concrete Test-Cases
### Contents

5.5 Our Methodology Overview and Tool Support .......................... 47  
5.5.1 Methodology Description ........................................ 47  
5.5.2 ViTAL: a Verification Tool for EAST-ADL Models using UPPAAL PORT ........................................ 48  
5.6 Validation of the Methodology on a Brake-by-Wire Use-Case ........ 50  
5.7 Research Goals Revisited ........................................... 50  
5.7.1 Paper A .................................................. 51  
5.7.2 Paper B .................................................. 51  
5.7.3 Paper C .................................................. 52  
5.7.4 Paper D .................................................. 52

6 Related Work ........................................................................ 53  
6.1 Analysis of Architectural Models of Embedded Systems ............ 53  
6.2 Testing Architectural Models starting from Requirements Speci- 
       fication ...................................................... 54

7 Conclusions and Future Work ............................................... 57  
Bibliography ............................................................................ 59

II Included Papers ..................................................................... 67

8 Paper A:  
A Methodology for Formal Analysis and Verification of EAST-ADL 
models ................................................................................. 69  
8.1 Introduction ................................................................. 71  
8.2 Preliminaries ............................................................... 72  
8.2.1 EAST-ADL ...................................................... 72  
8.2.2 UPPAAL PORT ................................................. 74  
8.3 Running Example: Brake-By-Wire .................................... 75  
8.4 Methodology and Proposed Solutions .................................. 76  
8.4.1 Architectural Mapping: EAST-ADL to INTERMEDIATE 
       COMPONENT MODEL ......................................... 77  
8.4.2 Integrating the Behavioral Formal Model: Mapping ICM to TA ...................................................... 78  
8.4.3 Simulation and Model Checking .................................. 80  
8.5 Tool-supported Modeling and Analysis ............................... 81  
8.5.1 Modeling Approach ................................................ 81  
8.5.2 Model transformation to UPPAAL PORT ................. 83
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.6</td>
<td>Case Study: Brake-by-Wire Control System</td>
<td>90</td>
</tr>
<tr>
<td>8.6.1</td>
<td>BBW System Model and Functionality</td>
<td>91</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Analysis and Verification</td>
<td>92</td>
</tr>
<tr>
<td>8.7</td>
<td>Related Work</td>
<td>94</td>
</tr>
<tr>
<td>8.8</td>
<td>Conclusion and Future Work</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>97</td>
</tr>
<tr>
<td>9</td>
<td>Paper B:</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Analyzing Industrial Architectural Models by Simulation and Model-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Checking</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>103</td>
</tr>
<tr>
<td>9.2</td>
<td>Brief Overview of the EAST-ADL Language</td>
<td>104</td>
</tr>
<tr>
<td>9.3</td>
<td>The Current Development Process in an Automotive Context</td>
<td>105</td>
</tr>
<tr>
<td>9.4</td>
<td>Our Methodology for Analyzing Architectural Models</td>
<td>106</td>
</tr>
<tr>
<td>9.5</td>
<td>An Example from Industry: Brake-by-Wire Case Study</td>
<td>108</td>
</tr>
<tr>
<td>9.6</td>
<td>Simulation of EAST-ADL Functional Architecture in Simulink</td>
<td>110</td>
</tr>
<tr>
<td>9.7</td>
<td>Formal Semantics of EAST-ADL as a network of Timed Automata</td>
<td>114</td>
</tr>
<tr>
<td>9.8</td>
<td>Analysis of EAST-ADL Models Using Model-Checking and</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>Statistical Model Checking</td>
<td></td>
</tr>
<tr>
<td>9.9</td>
<td>Related Work</td>
<td>120</td>
</tr>
<tr>
<td>9.10</td>
<td>Conclusions and Discussion</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>121</td>
</tr>
<tr>
<td>10</td>
<td>Paper C:</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>A Research Overview of Tool-Supported Model-based Testing of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Requirements-based Designs</td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>127</td>
</tr>
<tr>
<td>10.2</td>
<td>The Generic Model-based Testing Approach</td>
<td>129</td>
</tr>
<tr>
<td>10.3</td>
<td>Proposed Taxonomy Dimensions</td>
<td>131</td>
</tr>
<tr>
<td>10.3.1</td>
<td>The modeling notation</td>
<td>132</td>
</tr>
<tr>
<td>10.3.2</td>
<td>The test artifact</td>
<td>133</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Test selection criteria</td>
<td>134</td>
</tr>
<tr>
<td>10.3.4</td>
<td>The test generation method</td>
<td>135</td>
</tr>
<tr>
<td>10.3.5</td>
<td>The technology</td>
<td>136</td>
</tr>
<tr>
<td>10.3.6</td>
<td>The mapping</td>
<td>137</td>
</tr>
<tr>
<td>10.4</td>
<td>A Research Review of Model-based Testing Tools</td>
<td>137</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Selection criteria and procedures for including/excluding model-based</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>testing tools</td>
<td></td>
</tr>
<tr>
<td>10.4.2 Our Taxonomy</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>10.5 Running Example: The Coffee/Tea Vending Machine</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>10.6 Model-based Testing Tools for Pre/Post Notations</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>10.6.1 The Z language</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>10.6.2 The B-method</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>10.6.3 Spec#</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>10.6.4 AsmL</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>10.6.5 The Coffee/Tea Vending Machine in ProTest</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>10.7 Model-based Testing Tools for Transition-based Notations</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>10.7.1 Finite State Machines</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>10.7.2 Labeled Transition Systems</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>10.7.3 Timed Automata</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>10.7.4 UML statecharts.</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>10.7.5 The Coffee/Tea Vending Machine in UPPAAL CoVer</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>10.8 Model-based Testing Tools for Stochastic Models</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>10.8.1 Markov Chains</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>10.8.2 The Coffee/Tea Vending Machine in MaTeLo</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>10.9 Model-based Testing Tools for Data-Flow Models</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>10.9.1 Simulink, Lustre and Function Block Diagram</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>10.9.2 The Coffee/Tea Vending Machine in CompleteTest</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>10.10 Results and Discussion</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>10.11 Conclusions</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>171</td>
<td></td>
</tr>
</tbody>
</table>

11 Paper D:
A Model-Based Testing Framework for Automotive Embedded Systems 181
| 11.1 Introduction                      | 183 |
| 11.2 Preliminaries                     | 184 |
| 11.2.1 ViTAL                           | 184 |
| 11.2.2 Farkle                          | 187 |
| 11.3 Brake-by-Wire Case Study: Functionality and Structure | 188 |
| 11.4 From EAST-ADL to Code Validation: | |
| Methodology Overview                  | 189 |
| 11.5 Implementation Activities        | 190 |
| 11.5.1 Executable Semantics of UPPAAL PORT TA | 190 |
| 11.5.2 Implementing the System Model  | 191 |
| 11.6 Testing Activities               | 193 |
| 11.6.1 Generation of Abstract Test-Cases in ViTAL | 193 |
11.6.2  Generation and Execution of Concrete Test-Cases in Farkle ........................................... 194
11.7  Brake- by- Wire Revisited: Applying the Methodology . . . 195
  11.7.1  Creating the formal model ................................. 195
  11.7.2  Code implementation ....................................... 196
  11.7.3  Testing goal .................................................. 197
  11.7.4  Abstract test-case generation ............................ 198
  11.7.5  Python scripts generation ............................... 199
  11.7.6  Conformance between the abstract test-case and the Python script ..................................... 199
11.8  Related Work .................................................... 201
11.9  Conclusions and Future Work ................................. 202
Bibliography .......................................................... 202
I

Thesis
Chapter 1

Introduction

Embedded systems are computer systems with dedicated functionality and integrated within a larger mechanical or electrical system. Nowadays, they are widely used in the automotive industry, where mechanical and hydraulic technologies are being replaced by such embedded systems that can implement highly complex functions (e.g., cruise control, automatic braking, stability control, etc.). Using embedded systems increases the complexity and heterogeneity of the entire automotive system. Modern passenger cars contain more electronic components and have more computation power than the Apollo spaceship that flew to the Moon and back [18]. In this context, the automotive industry is trying to adapt its development process to this level of complexity, by moving towards model-based development and verification in order to manage the design intricacies. To support this claim, one can mention the use of the Simulink tool [14], which has become state-of-practice in the automotive industry, being equipped with modeling, simulation, and code generation capabilities. One other appealing solution towards model-based development is the use of architectural description languages that can be introduced earlier in the development process, to provide the well-defined system structure (as a set of interacting components), and capture related information such as timing properties and constraints, as well as component triggering annotations. Although there is solid research on the formal verification and model-based testing of complex functional models [34, 52, 49], only a few of the proposed methods are directly applicable to architectural descriptions, mostly due to the lack of support to formally specify and analyze the behavior of the functional components, as these are usually described in semi-formal languages such as UML.
Chapter 1

Introduction

Embedded systems are computer systems with dedicated functionality and integrated within a larger mechanical or electrical system. Nowadays, they are widely used in the automotive industry, where mechanical and hydraulic technologies are being replaced by such embedded systems that can implement highly complex functions (e.g., cruise control, automatic braking, stability control, etc.). Using embedded systems increases the complexity and heterogeneity of the entire automotive system. Modern passenger cars contain more electronic components and have more computation power than the Apollo spaceship that flew to the Moon and back [18]. In this context, the automotive industry is trying to adapt its development process to this level of complexity, by moving towards model-based development and verification in order to manage the design intricacies. To support this claim, one can mention the use of the Simulink tool [14], which has become state-of-practice in the automotive industry, being equipped with modeling, simulation, and code generation capabilities. One other appealing solution towards model-based development is the use of architectural description languages that can be introduced earlier in the development process, to provide the well-defined system structure (as a set of interacting components), and capture related information such as timing properties and constraints, as well as component triggering annotations. Although there is solid research on the formal verification and model-based testing of complex functional models [34, 52, 49], only a few of the proposed methods are directly applicable to architectural descriptions, mostly due to the lack of support to formally specify and analyze the behavior of the functional components, as these are usually described in semi-formal languages such as UML or
Simulink. To enjoy the fully-fledged advantages of reasoning, the architectural description languages should ideally be complemented by a component-aware analysis framework that is able to provide both formal verification and model-based testing capabilities.

The issues mentioned above have kindled our interest to introduce a methodology for formal verification and model-based testing of automotive embedded systems, assuming their architectural models as initial artifacts. We focus our work on EAST-ADL [16], an architectural description language dedicated to the development of automotive embedded systems and aligned with the AUTOSAR standard [29]. Since EAST-ADL lacks mathematically specified semantics, we start by providing formal semantics to the EAST-ADL architectural language [36]. Our chosen formalism is timed automata (TA), which enables functional and timing formal verification supported by a component-aware model-checker like UPPAAL PORT [31]. The later is an extension of UPPAAL for “read-execute-write” component-model semantics, which is the actual execution semantics assumed by EAST-ADL components.

Through a series of transformation, that we propose and implement, the EAST-ADL model extended with TA semantics is provided as input to the UPPAAL PORT model-checker, where one can simulate and check the model against (functional and timing) requirements, by exhaustively exploring all the possible interleavings of the function blocks [26, 36]. Since the state space explosion is a real problem when analyzing industrial-scale systems, we add to our analysis framework a new transformation that maps the elements of the EAST-ADL functional model intro a network of UPPAAL timed automata. By manually annotating the network of TA with stochastic behavior, we enable formal verification with UPPAAL SMC [19], within our analysis framework. UPPAAL SMC is an extension of UPPAAL for statistical model-checking, to verify quantitative properties by estimating the probabilities and probability distributions over time, with given confidence levels. The symbolic and statistical techniques complement each other: SMC can show results only up to a specified level of confidence and never for certain like symbolic techniques, yet it is a cheap way to generate and confirm safety counter-examples where symbolic techniques may employ expensive over-approximations [23]. However, SMC per se is not really in our focus, but rather the transformation from EAST-ADL to UPPAAL TA networks that facilitates the applicability of statistical model-checking as a complement to the symbolic one.

Testing, the main verification technique used by industry today, aims at gaining confidence in the software system through fault detection, i.e., observing the differences between the behavior of the implementation and the
expected behavior described in the specifications. Testing activities are time and resource consuming, and are often conducted by employing ad hoc, error prone, and expensive techniques [43]. This has boosted the development of potentially more efficient testing techniques, like model-based testing [53], where test construction and test execution can be (partially) automated. To collect information with respect to possible needs and gaps of current model-based testing methods used by industry and academia, we overview the state-of-the-art of requirements-driven model-based testing that is supported by mature tools. As a result of our research, we extend the earlier proposed taxonomy [54] with the artifacts considered in the test-case generation process and the mapping between abstract and executable test-cases. Such artifacts are models of: (i) functional system behaviors, (ii) extra-functional system behaviors, as well as (iii) requirements realizations in form of architectural models. The crux of our overview is the fact that we pick representative tools from each category of our taxonomy and apply them on a simple example, in order to observe and compare their outputs.

Our findings have motivated us to introduce a methodology for model-based testing against functional requirements, starting from EAST-ADL architectural models. For this, we propose a definition of the executable semantics of EAST-ADL models enriched with UPPAAL PORT TA behavior, and we extend our formal verification framework to automatically generate and execute test-cases for the system implementation. We take advantage of the UPPAAL PORT model-checker’s ability to generate witness traces by conducting reachability verification of EAST-ADL high-level models whose behavior is expressed as TA networks. The output of the verification is in form of traces of the model’s execution that are in fact our abstract test-cases. However, these test-cases cannot be used as such to test the implemented code, so we transform them into Python scripts, which act as their executable counterparts. We check the feasibility of the generated abstract test-cases by actually running the executable test-cases on the implemented code, in an attempt to obtain a pass or fail verdict.

We have implemented the above methodology in a tool called ViTAL, which integrates model-checking techniques with EAST-ADL models. The ViTAL tool has been paired with the Farkle tool [22], to enable automatic test-case conversion and execution on the system implementation. The tool-chain has been applied on an industrial use-case, namely the Brake-by-Wire system, to show the applicability of the presented methodology on an industrial-size automotive embedded system.
1.1 Thesis overview

The thesis is divided into two parts. The first part is a summary of our research, which includes: a short description of the preliminaries in Chapter 2, the main research goal of the thesis and its division into smaller and more manageable research goals in Chapter 4, a brief description of our contributions in Chapter 5, the research method used in our work in Chapter 3, a discussion on related works in Chapter 6, and conclusions and insights into our plans for future work in Chapter 7.

The second part of the thesis is a collection of papers that present the detailed solutions to our research goals. The following four papers are included in the second part of the thesis:


**Abstract:** The architectural design of embedded software has a direct impact on the final implementation, with respect to performance and other quality attributes. Therefore, guaranteeing that an architectural model meets the specified requirements is beneficial for detecting software flaws early in the development process. In this paper, we present a formal modeling and verification methodology for safety-critical automotive products that are originally described in the domain-specific architectural language EAST-ADL. We propose a model-based approach that integrates the architectural models with component-aware model-checking, and describe its tool support called ViTAL. The functional and timing behavior of each function block in the EAST-ADL model, as well as the interactions between function blocks are formally captured and expressed as Timed Automata models, which have precise semantics and can be formally verified with ViTAL. Furthermore, we show how our approach, supported by ViTAL, can be used to formally prove that the EAST-ADL system model fulfills the specified real-time requirements and behavioral constraints. We demonstrate that the approach improves the modeling and verification capability of EAST-ADL and identifies dependencies, as well as potential conflicts between different automotive functions before implementation. The method is substantiated by verifying an automotive braking system model, with respect to particular functional and timing requirements.

**Contribution:** The first four authors are the main contributors of this paper. Together with Eun-Young Kang, Eduard Paul Enoiu, and Cristina Seceleanu, I
have contributed in describing the formal verification methodology and the ViTAL tool. I was also responsible for applying the tool on the industrial use-case and presenting the results. The last two authors have contributed with useful ideas and comments.


Abstract: The software architecture of any automotive system has to be decided well in advance of production, so it is very desirable to assess its quality in order to obtain quick indications of errors at early design phases. In this paper, we present a constellation of analysis techniques for architectural models described in EAST-ADL. The methods are complementary in terms of covering EAST-ADL model analysis against a rich set of requirements, and in terms of the varying degree of confidence in the provided guarantees. Based on the needs of the current model-driven development in a chosen automotive context, we propose three analysis techniques of EAST-ADL architectural models, in an attempt to tackle some of the exposed design needs: simulation of EAST-ADL functions in Simulink, model-checking EAST-ADL models with timed automata semantics, and statistical model-checking in UPPAAL, applied on an automatically generated network of timed automata. An industrial Brake-by-Wire prototype is the case study on which we show the potential of simulating EAST-ADL models in Simulink, model-checking downscale EAST-ADL models, as well statistical model-checking of full model versions, in order to tame verification scalability problems.

Contribution: I was the main driver of this paper. My contribution consists in developing and presenting the transformation between the EAST-ADL model and the network of UPPAAL timed automata, and applying it on the industrial use-case. Henrik Kaijser contributed with the transformation of EAST-ADL into Simulink, together with its application on the industrial use-case. Marius Mikučionis has manually extended the network of UPPAAL timed automata with stochastic semantics to perform statistical model-checking on the industrial use-case. The last three authors have contributed with useful ideas and comments.

Paper C. A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. Raluca Marinescu, Cristina Seceleanu, Hélène

Abstract: Software testing aims at gaining confidence in software products through fault detection, by observing the differences between the behavior of the implementation and the expected behavior described in the specification. Nowadays, testing is the main verification technique used in industry, being a time and resource consuming activity. This has boosted the development of potentially more efficient testing techniques, like model-based testing, where test creation and execution can be automated, using an abstract system model as input. In this paper, we provide an overview of the state-of-the-art in tool-supported model-based testing that start from requirements-based models, by presenting and classifying some of the most mature tools available at this moment. Our goal is to get a deeper insight into the state-of-the-art in this area, as well as to form a position with respect to possible needs and gaps in the current tools used by industry and academia, which need to be addressed in order to enhance to applicability of model-based testing techniques. To achieve this, we extend an existing taxonomy with: (i) the test artifact, representing the type of information encoded in the model for the purpose of testing (i.e., functional behavior, extra-functional behavior, or the architectural description), and (ii) the mapping of test-cases that describes ways of using the generated test-cases on the actual system under test. To provide further evidence of the inner-workings of different model-based testing tools, we select four representative tools (i.e, ProTest, UPPAAL Cover, MaTeLo, and CompleteTest) that we apply on a simple and illustrative Coffee Vending Machine example, to show the differences in modeling notations, test-case generation methods, and the produced test-cases.

Contribution: I was the main driver of this paper. My main contribution consists in reviewing the relevant literature and running the selected tools. Together with Cristina Seceleanu, I wrote the paper and presented the results. Hélène Le Guen has provided the description and the application of the MaTeLo tool, and Paul Pettersson had contributed with useful ideas and comments on the paper.


Abstract: Architectural models, such as those described in the EAST-ADL language, represent convenient abstractions to reason about automotive embedded software systems. To enjoy the fully-fledged advantages of reasoning,
EAST-ADL models could benefit from a component-aware analysis framework that provides, ideally, both verification and model-based test-case generation capabilities. While different verification techniques have been developed for architectural models, only a few target EAST-ADL. In this paper, we present a methodology for code validation, starting from EAST-ADL artifacts. The methodology relies on: (i) automated model-based test-case generation for functional requirements criteria based on the EAST-ADL model extended with timed automata semantics, and (ii) validation of system implementation by generating Python test scripts based on the abstract test-cases. The scripts represent concrete test-cases that are executable on the system implementation. We apply our methodology to analyze the ABS function implementation of the Brake-by-Wire system prototype.

**Contribution:** The main contributors for the paper are the first two authors, which have contributed in equal parts in developing and presenting the model-based testing methodology and the tool support. My contribution consists in: (i) implementing and presenting the abstract test-case generation framework, and applying it on the industrial use-case, and (ii) introducing the executable semantics of the EAST-ADL model extended with UPPAAL PORT TA behavior. Mehrdad Saadatmand has described the test conversion and execution on the system under test. The other authors have contributed with useful ideas and comments.\(^1\)

---

\(^1\)Alessio Bucaioni has developed the tool adaptors (compliant to the OSLC standard) for integrating ViTAL and Farkle, which have not been included in the paper.
Chapter 2

Preliminaries

In this chapter, we introduce the (basic) notions used throughout the thesis. We start by providing a short overview of the model-driven development process in Section 2.1. Next, we briefly describe the E AST-ADL architectural language in Section 2.2, after which we describe the formalisms and the associated model-checking tools used in our formal verification framework in Section 2.3. We close the chapter with an overview of the model-based testing process in Section 2.4.

2.1 Model-driven Development of Systems

The model-driven development is an emerging development paradigm for complex software systems, where the specification and implementation is based on high level artifacts that become part of the overall solution. In model-driven development, the artifacts (or the models) represent the required functionality and the overall architecture of the system. Such models are not bounded to the underlying implementation technology and are closer to the problem domain than most of the popular programming languages, which makes them easier to be specified, understood, and maintained than code-based implementations [6, 50]. To be useful and effective, the models need to have five key characteristics: abstraction (the model is a reduced rendering of the system), understandability (the model appeals to the intuition), accuracy (the model must be a faithful representation of the system), predictiveness (the model should be used to correctly predict the property of the system through simulation or for-
Chapter 2

Preliminaries

In this chapter we introduce the (basic) notions used throughout the thesis. We start by providing a short overview of the model-driven development process in Section 2.1. Next, we briefly describe the EAST-ADL architectural language in Section 2.2, after which we describe the formalisms and the associated model-checking tools used in our formal verification framework in Section 2.3. We close the chapter with an overview of the model-based testing process in Section 2.4.

2.1 Model-driven Development of Systems

The model-driven development is an emerging development paradigm for complex software systems, where the specification and implementation is based on high level artifacts that become part of the overall solution. In model-driven development, the artifacts (or the models) represent the required functionality and the overall architecture of the system. Such models are not bounded to the underlying implementation technology and are closer to the problem domain than most of the popular programming languages, which makes them easier to be specified, understood, and maintained than code-based implementations [6, 50]. To be useful and effective, the models need to have five key characteristics: abstraction (the model is a reduced rendering of the system), understandability (the model appeals to the intuition), accuracy (the model must be a faithful representation of the system), predictiveness (the model should be used to correctly predict the property of the system through simulation or for-
mal analysis), and inexpensiveness (the model must be significantly cheaper to construct and analyze than the system). To attain the full benefits of model-driven development, the model’s potential for automation needs to be exploited through:

- Automatically generating partial or complete code-based implementations of the system based on the models, where the modeling languages take the role of implementation languages, and
- Automatically verifying the correctness of the models for the presence of desirable properties and the absence of undesirable ones, through formal analysis or simulation.

To support the standardization of model-driven development, the Object Management Group (a consortium of software vendors and users from industry, government, and academia) has proposed the Model-Driven Architecture initiative [44]. The initiative offers a conceptual framework for defining a set of standards in support of model-driven development, where the UML standard, along with several other technologies related to modeling play a key role. However, a model-driven development process can be attained with non-UML modeling approaches that rely on the use of formal modeling and verification techniques.

2.2 Architectural Modeling of Automotive Systems: EAST-ADL

The introduction of software architectures [51] has further shifted the focus from code-based design to coarser-grained architectural elements and their inter-connected structure, as part of a model-driven development methodology. Even if there is no universal definition for software architectures, they can be seen as the level of design that involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In this context, architecture description languages act as design environments used to express the conceptual architecture of a system [41].
Chapter 2. Preliminaries

Mal analysis), and inexpensiveness (the model must be significantly cheaper to construct and analyze than the system). To attain the full benefits of model-driven development, the model's potential for automation needs to be exploited through:

• Automatically generating partial or complete code-based implementations of the system based on the models, where the modeling languages take the role of implementation languages, and

• Automatically verifying the correctness of the models for the presence of desirable properties and the absence of undesirable ones, through formal analysis or simulation.

To support the standardization of model-driven development, the Object Management Group (a consortium of software vendors and users from industry, government, and academia) has proposed the Model-Driven Architecture initiative [44]. The initiative offers a conceptual framework for defining a set of standards in support of model-driven development, where the UML standard, along with several other technologies related to modeling play a key role. However, a model-driven development process can be attained with non-UML modeling approaches that rely on the use of formal modeling and verification techniques.

2.2 Architectural Modeling of Automotive Systems: EAST-ADL

The introduction of software architectures [51] has further shifted the focus from code-based design to coarser-grained architectural elements and their interconnected structure, as part of a model-driven development methodology. Even if there is no universal definition for software architectures, they can be seen as the level of design that involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In this context, architecture description languages act as design environments used to express the conceptual architecture of a system [41].

Figure 2.1: The EAST-ADL model of the BBW system at Design Level.
A number of architecture description languages have been proposed for modeling software/systems architectures, both within particular domains, as well as general-purpose architecture modeling languages. Through our work, we focus on EAST-ADL [16], an emerging architecture description language dedicated to the automotive domain and aligned with the AUTOSAR standard [29]. EAST-ADL provides a standardized format that captures different aspects of the automotive electronic system, like vehicle features, functions, requirements, software and hardware components, etc. The functionality of the system is defined at four levels of abstraction (Vehicle Level, Analysis Level, Design Level, and Implementation Level), each with its own specific role. For instance, the Analysis Level provides an abstract functional representation of the architecture without prescribing a specific hardware topology, while the Design Level presents a detailed functional representation of the architecture, plus the allocation of these elements onto the hardware architecture. The functional structure of the system at Analysis Level is represented by the Functional Analysis Architecture (FAA), while the structure of the system at Design Level is represented by the Functional Design Architecture (FDA) and the Hardware Design Architecture. Figure 2.1 depicts the Functional Design Architecture of a braking system without any mechanical connection between the brake pedal and four pedal actuators, and equipped with an ABS function, called the Brake-by-Wire use-case.

Each of these three system architectures rely on the definition of a set of FunctionTypes, representing abstract function component types that are used when modeling the functional structure of the system. Each EAST-ADL FunctionType consists of: (i) a set of FlowPorts to receive and provide data, (ii) a FunctionTrigger, which can be time-based or event-based, and (iii) a FunctionBehavior that is defined using external notations and tools (e.g. Simulink). For example, for the system depicted in Figure 2.1, the ABS FunctionType is periodically triggered every 10 ms based on the associated ABS_Periodic_Constraint, has three input data ports (RequestedTorqueIn, VehicleSpeedIn, and Wheel-SpeedIn), and has one output data port (ASBrakeTorqueOut). The system is modeled as a set of interconnected FunctionPrototypes, where each FunctionPrototype is an instantiation of the corresponding FunctionType. In this case, the ABS FunctionType is instantiated as the pABS_FL, the pABS_FR, the pABS_RL, and the pABS_RR FunctionPrototype. The execution of each FunctionPrototype is done based on the “read-execute-write” semantics, which enables semantically sound analysis and behavioral composition, and makes the function execution independent of the notation used when defining its internal behavior.
In EAST-ADL, the system is complete at each abstraction level, and structural elements of the system can be extended with annotations for orthogonal aspects like requirements, timing properties, generic constraints, etc. EAST-ADL provides a functional representation of the system, by presenting the interface of the components in the system and their interconnections, while the behavioral representation of the system is the set of FunctionBehaviors, most often specified through external tools, like Simulink. The representation of the actual implementation is not done in EAST-ADL, but rather in AUTOSAR, by providing traceability between the lower levels of EAST-ADL and AUTOSAR.

The Brake-by-Wire Use-case. The Brake-by-Wire (BBW) use-case is a braking system equipped with an ABS function, and without any mechanical connectors between the brake pedal and the brake actuators. A sensor attached to the brake pedal reads its position, which is used to compute the desired global brake torque. At each wheel, a sensor measures the speed of the respective wheel, which is used by the ABS algorithm together with the brake torque and the estimated vehicle speed to compute the actual brake torque that will be sent to the actuator. The ABS algorithm computes the slip rate based on the following equation:

\[
slipRate = \frac{vehicleSpeed - wheelSpeed \times Radius}{vehicleSpeed}
\]  

(2.1)

where \(vehicleSpeed\) is the speed of the vehicle, \(wheelSpeed\) is the speed of the wheel, and \(Radius\) is the radius of the wheel. The friction coefficient has a nonlinear relationship with the slip rate: when \(slipRate\) starts increasing, the friction coefficient also increases, and it reaches its peak value when \(slipRate\) is around 0.2. After that, further increase in \(slipRate\) reduces the friction coefficient of the wheel. For this reason, if \(slipRate\) is greater than 0.2 the brake actuator is released and no brake is applied, or otherwise the requested brake torque is used.

In Figure 2.1, we present the BBW system model in EAST-ADL, at Design Level, with annotations for timing properties like triggering period and execution time. The functionality of the system is realized by: (i) sensors (e.g., BrakePedalSensor), (ii) actuators (e.g., BrakeActuator), and (iii) computational blocks (e.g., BrakeTorqueMap, GlobalBrakeController, VehicleSpeedEstimator, and ABS). Most of the FunctionPrototypes in the system are time-triggered based on their period Period_Constraint. For instance, each of the four ABS FunctionPrototypes are triggered every 10ms, and their execution takes at most 2ms according to the associated ExecTime_Constraint. All the sensors and the actuators are event-triggered, meaning that their ex-
execution starts when they receive new data on the input ports. The \textit{pVehicle-SpeedEstimator} is also an event-triggered \textit{FunctionPrototype}, which is triggered when all the four input ports have received new data from the corresponding \textit{pLDM_Sensors}.

The BBW system has a set of requirements that need to be verified in order to ensure that they are met already at the architectural level. Here, we present two such requirements, in natural language:

- **Functional requirement:**
  \textit{If VehicleSpeedIn} $>$ \textit{ABSVehicleSpeedThreshold} and \textit{slip rate} $>$ \textit{ABSSlipRateThreshold}, then \textit{ABSBreakToqueOut} shall be set to \textit{0Nm}.

- **Timing requirement:**
  \textit{The time between the point when pBrakePedalSensor is triggered and when pGlobalBrakeController provides new data on the output ports should not exceed 130 ms.}

### 2.3 Formal Analysis by Model-checking

Formal analysis of system models is the process of rigorously exploring the behavior of the model expressed in an abstract mathematical notation, in order to get insights into its correctness (that is, meeting the defined requirements). A popular formal analysis technique is model-checking, which relies on efficient algorithms when deciding temporal logic formulas on finite state models. The main advantage of mode-checking based verification is that it provides exhaustive exploration of the (symbolic) states of the model with a high degree of automation.

Figure 2.2 provides a schematic overview of the model-checking technique. Given a formal model of the system (e.g., finite state machine), a model-checker can be used to verify whether the model satisfies a given system requirement formalized, for example, as a temporal logic formula. The model-checker reduces the verification to a reachability problem or to a temporal logic verification by algorithmically transversing the state transition of the formal model, and ensuring that the temporal logic formula is satisfied by each of the states of the model. The model-checker returns a Yes/No answer, and possibly a witness trace of the system’s execution (e.g., in case the formalized requirement is not satisfied, the model-checker can also provide a counter-example). In our work, we employ the UPPAAL model-checker [7] and its variants (UPPAAL PORT [31] and UPPAAL SMC [19]), which verify finite timed automata
Chapter 2. Preliminaries

Preliminary execution starts when they receive new data on the input ports. The $pVehicleSpeedEstimator$ is also an event-triggered FunctionPrototype, which is triggered when all the four input ports have received new data from the corresponding $pLDM Sensors$.

The BBW system has a set of requirements that need to be verified in order to ensure that they are met already at the architectural level. Here, we present two such requirements, in natural language:

- **Functional requirement:**
  
  If $VehicleSpeedIn > ABSVehicleSpeedThreshhold$ and $slip rate > ABSSlipRateThreshhold$, then $ABSBrakeTorqueOut$ shall be set to 0Nm.

- **Timing requirement:**
  
  The time between the point when $pBrakePedalSensor$ is triggered and when $pGlobalBrakeController$ provides new data on the output ports should not exceed 130 ms.

2.3 Formal Analysis by Model-checking

Formal analysis of system models is the process of rigorously exploring the behavior of the model expressed in an abstract mathematical notation, in order to get insights into its correctness (that is, meeting the defined requirements). A popular formal analysis technique is model-checking, which relies on efficient algorithms when deciding temporal logic formulas on finite state models. The main advantage of mode-checking based verification is that it provides exhaustive exploration of the (symbolic) states of the model with a high degree of automation.

Figure 2.2 provides a schematic overview of the model-checking technique. Given a formal model of the system (e.g., finite state machine), a model-checker can be used to verify whether the model satisfies a given system requirement formalized, for example, as a temporal logic formula. The model-checker reduces the verification to a reachability problem or to a temporal logic verification by algorithmically transversing the state transition of the formal model, and ensuring that the temporal logic formula is satisfied by each of the states of the model. The model-checker returns a Yes/No answer, and possibly a witness trace of the system’s execution (e.g., in case the formalized requirement is not satisfied, the model-checker can also provide a counter-example).

In our work, we employ the UPPAAL model-checker [7] and its variants (UPPAAL PORT [31] and UPPPAAL SMC [19]), which verify finite timed automata models [4] against requirements formalized in Timed Computation Tree Logic (TCTL) [3].

2.3.1 Timed Automata Models

The finite timed automata model was introduced by Alur and Dill [4] as a formalism for modeling real-time systems. The behavior is represented as a timed graph consisting of a finite set of locations and a finite set of labeled edges, and extended with real-valued timing constraints using real-valued clocks. The clock variables are initialized with zero when the system is started, and then increase synchronously with the same rate. Constraints on the clocks variables are represented as guards on edges and are used to restrict the behavior of the automaton, while actions and invariant conditions are used to enforce progress properties. To model concurrent systems, timed automata was extended with parallel composition that allows interleaving of actions as well as handshake synchronization.

The timed automata network is semantically defined as a labeled transition system, where the current state of the system is defined by the current location of all the timed automata in the network, together with the current valuation of all the variables. The transitions from one state to another can be: (i) discrete transitions, corresponding to taking an edge for which the guard is satisfied, or (ii) delay transitions where all clocks in the network are incremented with the same value.

We use a simple coffee vending machine to present different timed automata variants used by the UPPAAL model-checker, the UPPPAAL PORT model-checker - an extension of the UPPAAL tool for component-based systems,
Chapter 2. Preliminaries

and UPPAAL SMC - an extension of the UPPAAL tool for statistical model-checking. The specification of the coffee vending machine is simple: the user inserts three coins and presses the start button, and the coffee machine should serve the beverage within 15 seconds.

**UPPAAL PORT Timed Automata.** UPPAAL PORT [32] is a verification tool for component-based systems, and the formal model used in UPPAAL PORT is defined as a set of interconnected components, where each component has an interface and a timed behavior. The interface of each component consists of a set of input data ports, a set of output data ports, and a set of trigger ports. The data flow is defined via the data ports, while the control flow is defined through the trigger ports. The associated timed behavior is defined by the following tuple:

\[
B \triangleq \langle L, l_0, l_f, V_D, V_C, r_0, r_f, E, I \rangle
\]

where \( L \) is a finite set of locations, \( l_0 \) is the initial location, \( l_f \) is the final location, \( V_D \) and \( V_C \) are a set of data and clock variables, respectively, \( r_0 \) and \( r_f \) are sets of initial and final clock resets, \( E \) is a set of edges between locations, annotated with an action, a guard, and a set of clocks to be reseted, and \( I \) assigns clock invariants to locations.

A UPPAAL PORT component is initially idle, and it remains in this state until it is triggered, at which point the data variables are updated with the information from the ports. These variables are used during the execution phase, where the internal behavior of the component is executed. The execution phase ends with updating the data variables on the ports. UPPAAL PORT uses local variables and clocks, and data is passed from one component to the next via their respective ports.

Figure 2.3 depicts the Coffee Vending Machine modeled in UPPAAL PORT. Data ports, both input an output, are marked with a square, while the trigger ports are marked with a triangle. The system model is presented in Figure 2.3a and consists of two components: (i) a User Interface component, and (ii) a Coffee Maker component. When the User Interface is triggered, it reads the data from the data input port associated with the local variable \$coins, and the component executes the behavior according to the TA model. As depicted in Figure 2.3b, if the user has introduced three coins, then the guard \$coins == 3 holds and the transition from Entry to Exit is triggered. When the User Interface completes its execution, it triggers the execution of the Coffee Maker. The Coffee Maker produces the coffee within 5 to 15 time units, which is ensured by the invariant \( x <= 15 \) on the Entry location and the guard \( x >= 5 \) on the edge from Entry to Exit. When this edge is traversed, the \$serve_drink local
The following tuple: defined through the trigger ports. The associated timed behavior is defined by
consists of a set of input data ports, a set of output data ports, and a set of trig-
Figure 2.3b, if the user has introduced three coins, then the guard
Coffee Maker produces the coffee within 5 to 15 time units, which is ensured
variable is set to 1, and this data is provided to the data output port.

**UPPAAL Timed Automata.** The **UPPAAL TA** [7] is defined as a tuple:

\[
TA \equiv \langle L, l_0, C, A, E, I \rangle \tag{2.3}
\]

where \( L \) is a finite set of locations, \( l_0 \in L \) is the initial location, \( C \) is a set
of clocks, \( A \) is a set of possible actions, co-actions, and internal \( \tau \)-actions, \( E \) is a set of edges between locations with an action, a guard, and a set of
variables and clocks, and data is passed from one component to the next via
locations, annotated with an action, a guard, and a set of clocks to be reseted,
are sets of initial and final clock resets, \( f \) is a finite set of locations, \( E \) is a set of edges between
is modeled through synchronization channels (e.g., \( \text{channel!} \) and \( \text{channel?} \))
with rendezvous or broadcast semantics.
Figure 2.4 depicts the Coffee Vending Machine modeled as a network of two UPPAAL TA: the User TA depicted in Figure 2.4a, and the Coffee Maker TA depicted in Figure 2.4b. The user can insert a number of coins, which is modeled as an edge on which the coin variable is increased. Next, the user presses the button modeled as a synchronization channel named button that triggers the location change from Init to Prepare in the Coffee Maker TA, provided that the user has inserted three coins such that the guard \( coin == 3 \) evaluates to true. The Coffee Maker TA can be in the in the Prepare location between 5 and 15 time units, which is modeled as an invariant \( x<=15 \) and a guard \( x>=5 \) on the clock \( x \). The location change to Init in the Coffee Maker TA , also triggers the location change to Init in the User TA based on the reset synchronization channel.

**UPPAAL SMC Timed Automata.** To enable statistical model-checking with UPPAAL SMC [19], the model is viewed as a network of Priced Timed Automata (PTA), which are timed automata with clocks that can evolve at different (integer) rates in different locations. UPPAAL SMC allows the user to specify an arbitrary rate for the clock on any of the locations, and supports branching edges where weights can be added to give a distribution on discrete transitions. These rates and weights may be more than simple constants, they can be general expressions that depend on the states of the system. Simple arithmetic operations can be implemented on the clocks as floating point variables to allow the tool to compute nontrivial functions using small step integration. For more details and example on the modeling formalism used by UPPAAL SMC, we refer to reader to the work of Bulychev et al. [19].
2.3 Formal Analysis by Model-checking

2.3.2 Model-checking of Timed Automata Descriptions

The UPPAAL PORT Model-checker. UPPAAL PORT is an extension of the UPPAAL tool and supports simulation and model-checking of component-based systems modeled in the SaveCCM language [21]. In SaveCCM the system is defined as a set of interconnected components with explicitly defined input and output ports for data and control flow (see Figure 2.3a). The functional and timing behavior of each component is defined as a timed automaton, respectively. UPPAAL PORT takes as input the model described above, to provide formal verification of the system without the usual flattening of the TA network [31]. This is complemented by the Partial Order Reduction Technique (PORT), which tries to improve the efficiency of the analysis by exploring only a relevant subset of the state-space when model-checking. The tool also uses local time semantics [8] to increase independence, being suited for the analysis of “read-execute-write” component models.

The UPPAAL and UPPAAL SMC Model-checkers. UPPAAL is a model-checking tool for verifying timed systems based on an extended TA network formalism [7]. It uses symbolic semantics and symbolic reachability techniques to analyze dense-time state spaces against properties formalized as a subset of timed computation tree logic (TCTL). Recently, the modeling language has been extended with probabilistic and dynamical constructs, given a stochastic semantics of timed automata networks [24], and the tool has been equipped with statistical model-checking (SMC) algorithms [25] to decide qualitative properties in terms of probabilities and cost. The analytical reachability analysis of extended models is beyond decidability, hence the SMC engine generates stochastic simulations and employs statistical methods to estimate probabilities and probability distributions over time (and cost in general) with given confidence levels. The SMC results can then be interpreted as likely performance of the system, e.g., latency and energy consumption. The SMC algorithms use little memory (mostly to store statistics), do not suffer from state space explosion, the runtime complexity is independent from the model, and very often can be optimized even further using sequential methods in cases with consistent results. In practice, the symbolic and statistical techniques complement each other: SMC can show results only up to a specified level of confidence and never for certain like symbolic techniques, but it is a cheap way to generate and confirm safety counter examples where symbolic techniques may employ expensive over-approximation [23].
2.4 Model-based Testing

Testing is the verification process that aims at showing that the intended behavior of a system and the actual behavior of the implementations do not differ. Even though testing is the primary hardware and software verification technique used by industry today, it is often conducted in an ad hoc, error prone, and expensive manner. One possible way to overcome this is through model-based testing, which aims at automating the test-case generation and the execution on the system under test. The model-based testing process [54] is done in multiple steps, as depicted in Figure 2.5.

In model-based testing, the intended behavior of a system is most often expressed through requirements, specifications, and other similar documents, which are encoded into a formal model using explicit, behavioral notations (Step 1 in Figure 2.5). This model represents an abstraction of the system un-
2.4 Model-based Testing

Testing is the verification process that aims at showing that the intended behavior of a system and the actual behavior of the implementations do not differ. Even though testing is the primary hardware and software verification technique used by industry today, it is often conducted in an ad hoc, error prone, and expensive manner. One possible way to overcome this is through model-based testing, which aims at automating the test-case generation and the execution on the system under test. The model-based testing process \[54\] is done in multiple steps, as depicted in Figure 2.5.

In model-based testing, the intended behavior of a system is most often expressed through requirements, specifications, and other similar documents, which are encoded into a formal model using explicit, behavioral notations (Step 1 in Figure 2.5). This model represents an abstraction of the system under test, a smaller and simpler version of the actual system, and (most often) includes only the aspects of the system that are intended to be tested. Since there is no model notation that can satisfy all testing purposes, several modeling notations have been used by both academia and industry, for various purposes. Such examples include state-base notations like the Z language, transition-based notations like labeled transition system or timed automata, stochastic notations like Markov chains, etc.

Based on the requirements of the system, the test selection criteria defines the test-cases to be generated by specifying the features of the system to be tested prior to the actual test-case generation. Such a test selection criteria most often refers to a certain functionality of the system, to the structure of the model, or to different stochastic characteristics.

The formal model together with the test selection criteria are used to generate test-cases (Step 2 in Figure 2.5), most often represented as traces or pairs of inputs and expected outputs of the system. The test selection criteria is used to automatically generate the desired test-cases from an infinite number of possible tests. In most cases, these test are called abstract test-cases because they reflect only the information encoded in the formal model and are independent of the actual implementation, so they can not be directly executed on the system under test. Since the system model is an abstraction of the system under test, we need to add to the abstract test-cases all the details of the system implementation that were not mentioned in the abstract model (Step 3 in Figure 2.5). This can be done with the help of a transformation tool, or more often by writing an adaptor that wraps around the system under test and implements each abstract operation in term of the low-level system facilities.

The concrete test-cases are executed on the actual implementation of the system (Step 4 in Figure 2.5). At the end of each test execution, a verdict is returned (Step 5 in Figure 2.5), which represents the result of comparing the output produced by the system under test with the expected output provided by the test-case based on the formal model. The verdict can be pass, fail, or inconclusive (in some cases a decision cannot be made). For each failed test, the tester must determine the fault that has caused the failure. The fault can be either in the implementation or in the test itself.
Chapter 3
Research Method
Research methods are concrete ways to solve a given research problem, and a research methodology provides "a body of methods, rules, and postulates employed by a discipline" (Merriam-Webster dictionary). Such methods include [51, 56]: conceptual analysis, proof of concept (concept implementation), case study, data analysis, descriptive and exploratory surveys, etc. In this context, the research process describes the different stages for conducting research, where one or several research methods may be used in order to address a certain research goal. The relevant literature proposes fours major steps which, collectively, describe the cycle of the research process [35]. Figure 3.2 provides an overview of these four steps adapted to our particular computer engineering research context.

In order to tackle such a research process, different core and specific activities are required for each activity in the research process quadrant [35]. In Table 3.1, we detail some of the activities performed during our research process, tailored for the methodology proposed in Figure 3.2.

The ultimate goal of our research is to improve the state-of-the-art and state-of-practice in the analysis of architectural models like E AST-ADL. A major emphasis falls on using available techniques in the area of formal analysis and model-based testing that can provide frameworks that could be adopted by industry to evaluate our results through relevant activities. As depicted in Figure 3.2, our research has been triggered by the current problems and issues that the industry is facing (e.g., the adoption of future safety standards for passenger cars), together with the current solutions and gaps between the state-of-the-art and state-of-practice. To collect such information, we apply the
Chapter 3

Research Method

Research methods are concrete ways to solve a given research problem, and a research methodology provides “a body of methods, rules, and postulates employed by a discipline” (Merriam-Webster dictionary). Such methods include [51, 56]: conceptual analysis, proof of concept (concept implementation), case study, data analysis, descriptive and exploratory surveys, etc. In this context, the research process describes the different stages for conducting research, where one or several research methods may be used in order to address a certain research goal. The relevant literature proposes fours major steps which, collectively, describe the cycle of the research process [35]. Figure 3.2 provides an overview of these fours steps adapted to our particular computer engineering research context.

In order to tackle such a research process, different core and specific activities are required for each activity in the research process quadrant [35]. In Table 3.1, we detail some of the activities performed during our research process, tailored for the methodology proposed in Figure 3.2.

The ultimate goal of our research is to improve the state-of-the-art and state-of-practice in the analysis of architectural models like EAST-ADL. A major emphasis falls on using available techniques in the area of formal analysis and model-based testing that can provide frameworks that could be adopted by industry to evaluate our results through relevant activities. As depicted in Figure 3.2, our research has been triggered by the current problems and issues that the industry is facing (e.g., the adoption of future safety standards for passenger cars), together with the current solutions and gaps between the state-of-the-art and state-of-practice. To collect such information, we apply the
critical analysis of the literature and practice method from Table 3.1. Our research topic is formulated as a set of research goals focused on model-checking and model-based testing of architectural models used in the automotive industry. These research goals motivate us to provide a new framework for model-checking and model-based testing, tailored for automotive architectural languages, which could be adopted in the industrial practice. The direct result of our proposed framework and associated tool support is a series of research papers that present our work to the research community. This is achieved by
applying the proof-of-concept research method from Table 3.1. We apply the associated tool support on an industrial-scale case study to provide an initial evaluation of our work, and to evaluate the advantages and the limitations of the proposed framework, as part of applying the validation method from Table 3.1. Such results impact the state-of-the-art in the analysis of automotive architectural models, and (hopefully) will help in adopting similar model-checking techniques in an industrial context.

Figure 3.2: Overview of our research process.
Chapter 4

Research Goals

4.1 Problem Description

In a typical software development process, the system is implemented based on a set of informal, natural language requirements. In the automotive industry, these system requirements are decomposed and, usually, a model-based implementation of the resulted components is performed in dedicated tools like Simulink. Simulink is equipped with modeling, simulation, and code generation capabilities, and it has become state-of-practice in the automotive industry. However, the Simulink tool does not support the exhaustive verification needed in order to truly ensure the correctness of the system. Another solution towards model-based development is the use of architectural description languages, like EAST-ADL, that provide a structured approach for the functional representation of complex automotive systems directly at system level, and allow the representation of the associated behavior based on external notations such as UML, Simulink etc. Applying rigorous verification techniques already at the architectural level provides an attempt to gain early phase indications of possible functional and extra-functional errors. To enjoy the fully-fledged advantages of reasoning, an EAST-ADL model, for instance, needs to be described in a formal notation (such as timed automata) that is amenable to automated verification and model-based test-case generation. Model-checking techniques are good candidates for automated verification, since they provide exhaustive exploration of the state-space for both functional and extra-functional requirements (like timing properties), and can also be used to automatically generate test-cases. Consequently, there is a strong need for modeling and analysis.
Chapter 4

Research Goals

4.1 Problem Description

In a typical software development process, the system is implemented based on a set of informal, natural language requirements. In the automotive industry, these system requirements are decomposed and, usually, a model-based implementation of the resulted components is performed in dedicated tools like Simulink. Simulink is equipped with modeling, simulation, and code generation capabilities, and it has become state-of-practice in the automotive industry. However, the Simulink tool does not support the exhaustive verification needed in order to truly ensure the correctness of the system. Another solution towards model-based development is the use of architectural description languages, like EAST-ADL, that provide a structured approach for the functional representation of complex automotive systems directly at system level, and allow the representation of the associated behavior based on external notations such as UML, Simulink etc. Applying rigorous verification techniques already at the architectural level provides an attempt to gain early phase indications of possible functional and extra-functional errors. To enjoy the fully-fledged advantages of reasoning, an EAST-ADL model, for instance, needs to be described in a formal notation (such a timed automata) that is amenable to automated verification and model-based test-case generation. Model-checking techniques are good candidates for automated verification, since they provide exhaustive exploration of the state-space for both functional and extra-functional requirements (like timing properties), and can also be used to automatically generate test-cases. Consequently, there is a strong need for modeling and analysis
frameworks with a formal support that targeted the EAST-ADL models used in the automotive industry.

## 4.2 Research Goals

All the issues presented in Section 4.1 have kindled our motivation to develop an integrated framework that provides both formal verification and model-based testing for architectural languages (such as EAST-ADL) used in the automotive industry. Below we define our main goal, as follows:

**Overall Goal.** To investigate how model-checking techniques can be used for both the analysis and model-based testing of automotive embedded systems, starting from system-level architectural models.

Since the overall goal is both large and too abstract to be directly addressed, we have divided it into five concrete and more manageable goals. In order to provide a verification framework tailored for EAST-ADL architectural models, one needs to create a formal model that integrates both the functional and the behavioral representation of the system. Hence, we have formulated the first goal as follows:

**RG 1.** Define a formal semantics for the EAST-ADL functional architecture.

The defined formal semantics provides a basis for the formal representation of the EAST-ADL models, which can be formally verified using model-checking techniques. This gives rise to our second goal as follows:

**RG 2.** Develop a methodology for model-checking EAST-ADL models extended with formal semantics.

Since testing is the “de facto” verification technique used by the industry, it is highly desirable to generate test-cases automatically, in an attempt to increase the efficiency of the testing process. For this, we first look at the state-of-the-art in model-based testing of requirements-based system specification, in order to identify trends as well as current needs and gaps. Therefore, we have formulated our third goal as follows:

**RG 3.** Identify the differences and common characteristics between existing techniques for requirements-based model-based testing of embedded systems.

Once we have investigated the relevant related works regarding requirement-based model-based testing of embedded systems, we need to extend our formal
4.2 Research Goals

analysis framework with model-based testing features. This gives rise to our fourth goal, as follows:

**RG 4.** *Develop a methodology and tool support for integrating model-checking and model-based testing that starts from EAST-ADL descriptions.*

Last but not least, we want to check the applicability of our complete verification framework on a real-scale industrial system. Thus, our last goal is formulated as follows:

**RG 5.** *Validate the methodology on an industrial prototype.*
Chapter 5
Thesis Contribution

In this chapter, we outline the contributions and results of this thesis, which address the thesis goals described in Section 4.2. The main contribution of this thesis consists of a methodology based on model-checking techniques, for analyzing automotive systems described by their EAST-ADL architectural models. This thesis proposes solutions for the verification and code testing (by the automatic generation of abstract test-cases) through model-checking of the EAST-ADL models. The entire methodology has been applied on an industrial case-study, namely the Brake-by-Wire use-case, to show the applicability of our work on an industrial-size automotive embedded system. The BBW case-study on which we will briefly exemplify our contributions, has been introduced in Section 2.2.

5.1 Formal Verification of EAST-ADL Models

The automotive systems that we consider, implement real-time functions that make the verification of both functional and timely behavior a necessity. To provide formal verification of the system against its requirements, we need to create a formal model based on the architectural representation. Next, by employing formal verification techniques, such as model-checking, we can show that all the possible behaviors, represented by all the possible executions paths in the formal model, respect the requirement under verification. In order to achieve this, we start from the informal semantics of EAST-ADL , and we provide a formal model based on TA semantics.
Chapter 5

Thesis Contribution

In this chapter, we outline the contributions and results of this thesis, which address the thesis goals described in Section 4.2. The main contribution of this thesis consists of a methodology based on model-checking techniques, for analyzing automotive systems described by their EAST-ADL architectural models. This thesis proposes solutions for the verification and code testing (by the automatic generation of abstract test-cases) through model-checking of the EAST-ADL models. The entire methodology has been applied on an industrial case-study, namely the Brake-by-Wire use-case, to show the applicability of our work on an industrial-size automotive embedded system. The BBW case-study on which we will briefly exemplify our contributions, has been introduced in Section 2.2.

5.1 Formal Verification of EAST-ADL Models

The automotive systems that we consider, implement real-time functions that make the verification of both functional and timely behavior a necessity. To provide formal verification of the system against its requirements, we need to create a formal model based on the architectural representation. Next, by employing formal verification techniques, such as model-checking, we can show that all the possible behaviors, represented by all the possible executions paths in the formal model, respect the requirement under verification. In order to achieve this, we start from the informal semantics of EAST-ADL, and we provide a formal model based on TA semantics.
The timed automata formalism is a suited modeling notation for complex systems with timing behavior, and can be easily paired with a powerful model-checker like UPPAAL for formal verification. Our main contributions towards developing means of model-checking EAST-ADL models consist in a series of transformations that provide formal semantics to the EAST-ADL architectural model as networks of timed automata, as well as integrating in the same framework means for formal verification and model-based testing of such models, based on various extensions of the UPPAAL model-checker.

5.1.1 Formal Semantics of EAST-ADL as Timed Automata

In this section, we show how the informal semantics of the EAST-ADL architecture language can be captured formally, by networks of TA.

**UPPAAL PORT compatible TA semantics.** In UPPAAL PORT [31], each component is defined by its interface and its timed behavior. We take advantage of this and we propose a series of transformations that provide formal semantics to EAST-ADL, in order to create a formal model that is compatible to the input language of UPPAAL PORT. In our approach, the system model and the interface of the components are modeled in EAST-ADL, whereas each component has its behavior modeled as UPPAAL PORT compatible TA semantics.

To achieve this, we first map the elements of the EAST-ADL model to an Intermediate Component Model (ICM). The EAST-ADL component interface is defined in terms of EAST-ADL FunctionTypes, as the following tuple:

\[
FunctionType \triangleq \langle F_P, D_P, Con, TC \rangle, \tag{5.1}
\]

where \( F_P \) is a set of FunctionPrototypes \( f_p \)'s of a FunctionType, \( D_P \) is the set of data ports, defined as the reunion of input ports and output ports, \( Con \) is the set of connectors, and \( TC \) is the set of timing constraints. Similarly, the ICM can also be defined as a tuple, as follows:

\[
ICM \triangleq \langle Comp, P_{in}, P_{out}, trig, Connections, TC_{ICM} \rangle, \tag{5.2}
\]

where \( Comp \) is the set of components, \( Connections \) is the set of connections, \( P_{in} \), and \( P_{out} \) are the input and output data flow ports, respectively, \( trig \) is the trigger variable that is set to true by a clock component, and \( TC_{ICM} \) is the ICM’s set of timing constraints.

This mapping (between EAST-ADL FunctionType and ICM) is a function \( \pi : FunctionType \rightarrow ICM \), which maps each FunctionPrototypes \( f_p \) to an ICM component, input ports to the ICM’s component dataflow input ports,
output ports to the ICM’s component dataflow output ports, connectors to the ICM’s component connections, and the FunctionPrototypes’s timing constraints TC to ICM’s timing constraints, $T C_{ICM}$.

Next, we specify the behavior associated with each EAST-ADL FunctionPrototype by assigning an UPPAAL PORT TA to the corresponding ICM Comp. The TA encapsulates the internal behavior of the FunctionType, which in the EAST-ADL language is defined based on external notations and tools [16]. Figure 5.1 depicts such a possible behavior model associated with the pABS_FL FunctionPrototype, in terms of UPPAAL PORT TA. The behavior of the TA model is described as follows. First, the speed of the car ($v$) is evaluated: if the car has no speed then no brake force is applied ($\text{torqueABS} == 0$), otherwise the slip rate is evaluated. If the slip rate exceeds 0.2, no braking force should be applied to not block the wheel (again $\text{torqueABS} == 0$), otherwise the desired braking torque $\text{wheelABS}$ is sent to the corresponding actuator ($\text{torqueABS} == \text{wheelABS}$). In our TA model, we are evaluating $\text{slip}_{\text{rate}} > 0.2$ as $v < 5(v - w \times R)$ (based on Equation 2.1).

![Figure 5.1: The TA model associated with the pABS_FL FunctionPrototype.](image)

In order to integrate the two models, ICM and the set of TA behaviors, the UPPAAL PORT TA tuple introduced in Section 2.3.1 is extended as follows:

$$TA \triangleq \langle L \cup \{l_{\perp}\}, l_0, l_f, V_C, V_D, r_0, r_f, E, I \rangle, \quad (5.3)$$

where the set of TA locations $L$ is extended with the idle location $l_{\perp}$, representing the location of the TA that corresponds to FunctionPrototype not being active. The initial location is denoted by $l_0 \in L$ and the final location is denoted by $l_f \in L$, such that no edges in $E$ are leading into $l_0$ and from $l_f$, respectively. For this, we define the action $\text{Read}(P_{\text{in}})$ for reading variables from the input port and $\text{Write}(P_{\text{out}})$ for writing variables onto output ports. The other elements in the tuple are not changed. The resulting model, the ICM extended with timed automata, represents our formal model that is compatible
to the input language of UPPAAL PORT. This model obeys the same “read-
execute-write” semantics as the EAST-ADL language.

For the \textit{pABS\_FL FunctionPrototype} and the associated TA behavior (pre-

tended in Figure 5.1), the formal model compatible with the input language

of UPPAAL PORT assumes the following: (i) a mapping between the UPPAAL

PORT local variables and the EAST-ADL port names (see Figure 5.2) and (ii)

the behavior of the \textit{FunctionPrototype} extended with the \textit{idle} location and the

\textit{read} and \textit{write} actions (see Figure 5.3).

\begin{verbatim}
1 <MAPPING xstamodel="ABS\_FL.xsta">
2  <MAP var="wheelABS" port="RequestedTorque\_FL"/>
3  <MAP var="torqueABS" port="ABSTorque\_FL"/>
4  <MAP var="v" port="VehicleSpeed\_kmph\_FL"/>
5  <MAP var="w" port="WheelSpeed\_rpm\_FL"/>
6 </MAPPING>

Figure 5.2: Mapping between UPPAAL PORT local variables and the EAST-

ADL ports for the \textit{pABS\_FL}.

\begin{verbatim}
1 <MODEL type="uppaal:declarations">
2  int WheelSpeed\_rpm\_FL;
3  int RequestedTorque\_FL;
4  int ABSTorque\_FL;
5  int VehicleSpeed\_kmph\_FL;
6  int R=1;
7 </MODEL><MODEL type="uppaal:behaviour">
8  state idle, Entry, CalcSlipRate, Exit;
9  init idle;
10  trans idle->Entry { guard false; },
11    Entry->Exit { guard v==0; assign torqueABS=0;},
12    Entry->CalcSlipRate { guard v>0; },
13    CalcSlipRate->Exit { guard v>=5*(v-w*R); assign torqueABS=wheelABS; },
14    CalcSlipRate->Exit { guard v>5*(v-w*R); assign torqueABS=0; },
15    Exit->idle { guard false; };
16 </MODEL></BEHAVIOUR>

Figure 5.3: The extended behavior of the \textit{pABS\_FL FunctionPrototype}.

Figure 5.3 shows the possible states of the \textit{pABS\_FL TA}, extended with the

\textit{idle} state (line 2), which is also the initial state of the system (line 3). The

possible transitions of the TA are the \textit{read} action that is represent be the transition

from \textit{idle} to \textit{Entry} (line 4), the internal transitions of the TA model (lines
5.1 Formal Verification of EAST-ADL Models

37

5-15), and the write action that is represent by the transition from Exit to idle (line 16).

UPPAAL compatible TA semantics. The semantics of the EAST-ADL FunctionTypes as UPPAAL PORT lets one use the efficient UPPAAL PORT verifier to analyze the EAST-ADL models. However, in order to verify complex behaviors and scale the verification by statistical techniques, we have also proposed formal semantics to the EAST-ADL architectural language as traditional UPPAAL TA networks. For this, we have defined mapping rules for both the EAST-ADL FunctionType interface, as well as its associated internal behavior [39]. The mapping produces a network of two synchronized TA for each EAST-ADL FunctionPrototype, respectively:

- The Interface TA is based on the elements provided in the architectural model, and
- The Behavior TA needs to be manually edited to match the behavior of the corresponding FunctionPrototype.

Figure 5.4: The TA model for the pABS_FL EAST-ADL FP.
We present the principles of the transformation on two FunctionPrototypes from the BBW model, the time-triggered \texttt{pABS\_FL} and the event-triggered \texttt{pVehicleEstimator}. Figure 5.4 shows the UPPAAL TA model of the time-trigger \texttt{pABS\_FL FunctionPrototype} as a parallel composition of \texttt{Interface TA} (Figure 5.4a) and \texttt{Behavior TA} (Figure 5.4b). Each input and output port is represented as a global variable in the TA network. The \texttt{Interface TA} has four locations: (i) the initial \texttt{Idle} location, (ii) the \texttt{Read} location, (iii) the \texttt{Exec} location, and (iv) the \texttt{Write} location. The \texttt{Read} and \texttt{Write} locations have been defined as committed locations, which ensures that there cannot be any delays in these locations and the TA moves to the next location immediately. On the edge between \texttt{Idle} and \texttt{Read}, the input port variables are updated based on the connectors in the system, together with the triggering elements. For timed-triggered FunctionPrototypes the transformation produces a local clock \( x \), plus invariants and guards on TA, and the trigger is implemented by the invariant \( x \leq \text{period} \), which ensures that the TA can leave the \texttt{Idle} location only when the clock \( x \) reaches the value of the corresponding \texttt{period}, which is 10 time units (Figure 2.1). The edge between \texttt{Read} and \texttt{Exec} bears a synchronization channel that starts the execution of the \texttt{Behavior TA}, while the edge between \texttt{Exec} and \texttt{Write} has a synchronization channel that ends the execution of the \texttt{Behavior TA}. Concretely, once the TA has reached the \texttt{Read} location, the edge \texttt{Read to Exec} is traversed, and the synchronization channel \texttt{pABS\_FL\_beh\_start} triggers the edge from \texttt{Init to Beh} in the corresponding \texttt{Behavior TA} depicted in Figure 5.4b. The \texttt{Behavior TA} performs the desired computation of the brake torque (based on the slip rate) right away, and moves to the next location. The two TA stay in these locations until the clock \( x \) has reached the execution time \texttt{exec}, which has the value of 2, at which point the edge \texttt{Exec to Write} is traversed in the \texttt{Interface TA} and the synchronization channel \texttt{pABS\_FL\_beh\_stop} also takes the \texttt{Behavior TA} to the \texttt{Init} location. The edge between \texttt{Write} and \texttt{Idle} is dedicated to updating any necessary variables. Finally, the TA returns to the \texttt{Idle} location, and remains there until the component is triggered again.

Figure 5.5 presents the UPPAAL TA model of the \texttt{pVehicleSpeedEstimator FunctionPrototype}. Figure 5.5a presents the \texttt{Interface TA} associated with this FunctionPrototype, which behaves in the similar manner. The only difference is that \texttt{pVehicleSpeedEstimator} is an event-triggered FunctionPrototype and the \texttt{Interface TA} is triggered only when new data arrives on all the four input ports. For each port, e.g., \texttt{pVehicleSpeedEstimator\_WheelSpeed\_FLIn}, the transformation provides a data variable, e.g., \texttt{pVehicleSpeedEstimator\_WheelSpeed\_FLIn}, and a trigger, e.g., \texttt{pVehicleSpeedEstimator\_WheelSpeed\_FLIn\_trig} that is set to 1 when the triggering component ends its execution.
We present the principles of the transformation on two FunctionPrototypes from the BBW model, the time-triggered pABS FL and the event-triggered pVehicleEstimator. Figure 5.4 shows the UPPAAL TA model of the time-trigger pABS FL FunctionPrototype as a parallel composition of Interface TA (Figure 5.4a) and Behavior TA (Figure 5.4b). Each input and output port is represented as a global variable in the TA network. The Interface TA has four locations: (i) the initial Idle location, (ii) the Read location, (iii) the Exec location, and (iv) the Write location. The Read and Write locations have been defined as committed locations, which ensures that there cannot be any delays in these locations and the TA moves to the next location immediately. On the edge between Idle and Read, the input port variables are updated based on the connectors in the system, together with the triggering elements. For timed-triggered FunctionPrototypes the transformation produces a local clock $x$, plus invariants and guards on TA, and the trigger is implemented by the invariant $x < period$, which ensures that the TA can leave the Idle location only when the clock $x$ reaches the value of the corresponding period, which is 10 time units (Figure 2.1). The edge between Read and Exec bears a synchronization channel that starts the execution of the Behavior TA, while the edge between Exec and Write has a synchronization channel that ends the execution of the Behavior TA. Concretely, once the TA has reached the Read location, the edge Read to Exec is traversed, and the synchronization channel pABS FL behstart triggers the edge from Init to Beh in the corresponding Behavior TA depicted in Figure 5.4b. The Behavior TA performs the desired computation of the brake torque (based on the slip rate) right away, and moves to the next location. The two TA stay in these locations until the clock $x$ has reached the execution time exec, which has the value of 2, at which point the edge Exec to Write is traversed in the Interface TA and the synchronization channel pABSFL behstop also takes the Behavior TA to the Init location. The edge between Write and Idle is dedicated to updating any necessary variables. Finally, the TA returns to the Idle location, and remains there until the component is triggered again.

Figure 5.5 presents the UPPAAL TA model of the pVehicleSpeedEstimator FunctionPrototype. Figure 5.5a presents the Interface TA associated with this FunctionPrototype, which behaves in the similar manner. The only difference is that pVehicleSpeedEstimator is an event-triggered FunctionPrototype and the Interface TA is triggered only when new data arrives on all the four input ports. For each port, e.g., pVehicleSpeedEstimator WheelSpeed FLIn, the transformation provides a data variable, e.g., pVehicleSpeedEstimator WheelSpeed FLIn, and a trigger, e.g., pVehicleSpeedEstimator WheelSpeed FLIn trig that is set to 1 when the triggering component ends its execution.

The result of the transformation is a formal model compatible with the input model of the UPPAAL tool. For an EAST-ADL model consisting of $n$ FunctionPrototypes, the corresponding formal model consists of a network of $2 \times n$ TA with $2 \times n$ synchronization channels, $n$ local clocks, and global variables for each input and output port in the EAST-ADL model. The Interface TA contains the elements of the corresponding FunctionPrototype, while the Behavior TA needs to be manually edited to implement the desired functionality (see Figure 5.4b and 5.5b).

5.1.2 Model-checking EAST-ADL as Timed Automata Networks

The main purpose of the two transformations presented in Section 5.1.1 is to provide a formal semantics to EAST-ADL FunctionType, which can be used in
the formal verification of automotive embedded systems described in EAST-ADL. We investigate the analysis of the chosen architectural model with the following model-checkers: (i) UPAAAL, a well-known model-checker for real-time systems, (ii) UPAAAL PORT, an extension of UPAAAL for component-based systems, and (iii) UPAAAL SMC, an extension of the UPAAAL tool for statistical model-checking.

Verification with UPAAAL PORT. The first transformation presented in Section 5.1.1 enables the component-based formal verification of the EAST-ADL models [26, 36]. UPAAAL PORT is an extension of the UPAAAL tool, which uses the Partial Order Reduction Technique (PORT) [32], to improve the efficiency of the model-checking analysis by using local time semantics to increase independence that allows the analysis of components in isolation followed by synchronization to a shared state, thus making the model-checker suited for the analysis of “read-execute-write” component models (like in EAST-ADL).

For the BBW system presented in Figure 2.1, the state space explodes during verification. On a computer with 1.8 Ghz Intel processor and 8GB memory, we could verify only a simplified version with two wheels. In order to verify the requirements, we first encode them in TCTL, as follows:

- Functional requirement:
  \[ \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \neg \ne
Chapter 5. Thesis Contribution

The formal verification of automotive embedded systems described in EAS-TADL. We investigate the analysis of the chosen architectural model with the following model-checkers: (i) UP-PAAL, a well-known model-checker for real-time systems, (ii) UP-PAAL PORT, an extension of UP-PAAL for component-based systems, and (iii) UP-PAAL SMC, an extension of the UP-PAAL tool for statistical model-checking.

Verification with UP-PAAL PORT. The first transformation presented in Section 5.1.1 enables the component-based formal verification of the EAS-TADL models [26, 36]. UP-PAAL PORT is an extension of the UP-PAAL tool, which uses the Partial Order Reduction Technique (PORT) [32], to improve the efficiency of the model-checking analysis by using local time semantics to increase independence that allows the analysis of components in isolation followed by synchronization to a shared state, thus making the model-checker suited for the analysis of “read-execute-write” component models (like in EAS-TADL).

For the BBW system presented in Figure 2.1, the state space explodes during verification. On a computer with 1.8 Ghz Intel processor and 8GB memory, we could verify only a simplified version with two wheels. In order to verify the requirements, we first encode them in TCTL, as follows:

- Functional requirement: $A[pABS_FL.v < 5 \times (pABS_FL.v - pABS_FL.w \times pABS_FL.R)] \implies pABS_FL.torqueABS == 0$.
  The requirement holds.

- Timing requirement: Assuming a TA model with timing annotations, different timing properties (like end-to-end deadlines) can be verified by implementing a new Comp in the ICM that acts as a monitor. Such a monitor could be triggered when new data is provided to the input port of the pBrakePedalSensor, and its local clock could be used to measure the execution time of the system until pGlobalBrakeController provides new data on the output ports, in order to verify the end-to-end deadline for the components assigned to the brake pedal.

Verification with UP-PAAL. The formal model based on UP-PAAL TA semantics can be easily verified with the UP-PAAL model-checker [39]. The transformation enables symbolic simulation and formal verification of system requirements formalized as TCTL properties. However, due to the possible large number of TA in the network, coupled with the large number of clocks and variables, the state-space explosion can become a real problem.

Verification with UP-PAAL SMC. In an attempt to overcome the state-space explosion problem, we employ statistical model-checking techniques to generate stochastic simulations, and use statistical methods to estimate probabilities and probability distributions over time with given confidence levels [39]. However, currently we achieve this by manually adding probabilistic extensions to the network of TA of the four-wheels BBW model, in order to investigate other types of properties, like the latency between the wheel sensor and the brake pedal actuator [40].

5.2 Tools for Model-based Testing: an Overview

Model-based testing tools are relying on different characteristics that should be used to understand the advantages and limitations involved in integrating a particular model-based testing tools into the software development process. In 2012, Utting et al. [54] have devised a taxonomy that identified different dimensions of model-based testing. From this taxonomy, we appropriate four
of the proposed dimensions into our proposed taxonomy, namely the model notations, the test selection criteria, the test generation method, and the text execution. We extend this taxonomy by proposing two new dimensions as shown in Figure 5.7: (i) the test artifact represented by the model and (ii) the mapping support between abstract and executable tests. The test artifact represents the type of information encoded in the model for the purpose of testing, namely the functional behavior, extra-functional behavior, or the architectural description. The test-case generation process, relies on the definition of a formal model, which is used to automate the test-case generation. Consequently, the result of the test-case generation process can be a set of abstract test-cases, which
encode only the information from the system model and can not be directly run on the SUT, or executable test-cases, if the model-based testing tool incorporates transformation rules that connect the model and the implementation of the system under test.

Figure 5.8: Model-based testing tools by the type of model notation supported

Our goal is to provide an overview of the state-of-the-art in model-based testing by presenting existing model-based testing tools, and classifying them based on the proposed taxonomy (see Figure 5.7). Due to the large number of models proposed by researchers, we focus only on model-based testing techniques supported by an integrated tool, which handles modeling, test-case generation, test-case selection and possibly execution. Our inclusion criteria has provided us with 33 tools, and part of our results are presented in Figure 5.8. As shown in Figure 5.8 a large majority of model-based testing tools can support transition-based notations (23) and pre/post notations (5). Just a few tools can handle stochastic (2) and data-flow notations (3).

5.3 Implementation guidelines for EAST-ADL+TA Models

The implementation is an important, labor intensive, and error prone phase in the development process of any software system. To ease the implementation process, we provide a set of guidelines for the implementation of the
EAST-ADL model extended with TA semantics as C code [40]. The guidelines include: (i) a definition for the executable semantics of the UPPAAL PORT TA, and (ii) implementation rules for the system model.

**Executable Semantics of UPPAAL PORT TA.** The TA behavior of an EAST-ADL FunctionPrototype can be non-deterministic, and, in order to obtain an implementation of the EAST-ADL model, we need to define the deterministic semantics of the model. For this, we adapt the UPPAAL PORT TA to the approach proposed by Amnell et. al [5] for task automata code synthesis. The non-determinism of the semantic representation of the UPPAAL PORT TA stems from the possible transitions in the TA model (read actions, write actions, internal transitions, or time-delays). Based on the previous work on TA [5], we resolve non-determinism, as follows:

- Action non-determinism is resolved by implementing a function which assigns unique priorities to each edge in the UPPAAL PORT TA to establishes the order in which the transitions are fired. If several transitions are enabled, the transition with the highest priority is fired;

- Time non-determinism is resolved by implementing the maximal progress assumption [55], in which delay transitions are forbidden if an action transition is enabled. This means that the TA should fire all the enabled transitions until no enabled transition exists anymore.

The above solutions provides deterministic semantics for the behavior defined by the UPPAAL PORT TA model, and ensures conformance of the deterministic implementation to the high-level behavioral model (all the transition sequences possible in the implementation model are also possible in the original one), thus guaranteeing preservation of the safety properties of the EAST-ADL behavioral TA model.

**Implementing the System Model.** We approach the problem of code implementation as a mapping activity between the EAST-ADL system model extended with TA semantics and C code. For the implementation guidelines, we refer the reader to our recent work [40], included later in this thesis. Figure 5.9 presents an example of such an implementation for the pABS_FL FunctionPrototype presented in Figure 5.1.
5.4 Model-based Testing Starting from EAST-ADL

Besides using exhaustive formal analysis to ensure that the architectural model respects its functional and timing requirements, we need to check that the system’s implementation conforms with the above mentioned requirements. For this, we extend our verification framework with test-case generation capabilities [40]. Our methodology relies on: (i) employing the UPPAAL PORT model-checker for automated model-based test-case generation based on the EAST-ADL model extended with TA semantics, and (ii) validating the system implementation by generating Python test scripts based on the abstract test-cases and executing them on the actual code to get a pass or fail verdict.

5.4.1 Generation of Abstract Test-Cases

We employ UPPAAL PORT to automate the abstract test-case generation, by exploiting the ability of the UPPAAL PORT model-checker to automatically gen-

```c
void mbatAbs_calc(MbatAbsInput* input, void* hdl)
{
    state = Idle;
    /* Internal variables of automaton */
    float s;
    /* Output variables */
    U32 TorqueABS=0;

    state = Entry;
    while(state != Exit) {
        switch(state) {
            case Entry: {
                if(input->v > 0) {state=CalcSlipRate; }
                else
                    if(input->v == 0) { TorqueABS=0; state=Exit; }  
                else { // Error }
                    break; }
            case CalcSlipRate: {
                s = ((float)(input->v-input->w*input->R)/input->v);
                if(s > 0.2) { TorqueABS=0; }
                else { TorqueABS=input->WheelABS; }
                state = Exit;
                break; }
            case Exit: { break; } }
        printf(" Tracing ABS calculation state:%d
", state);
        mbatAbs_transition(state, input->w, input->WheelABS,
            input->v, TorqueABS, input->R, (MBAT_TRC*)hdl);
        state = Idle; }
```

Figure 5.9: Implementation of the pABS_FL FunctionPrototype.
generate witness traces for reachability properties specified in TCTL. The model-checker requires as input the abstract formal model, which is represented by the EAST-ADL model extended with TA semantics, and the testing goal, which is represented by a functional requirement of the system (or a collection of such requirements) formalized as a TCTL reachability property. The model-checker returns a witness trace, which is a sequence of states and transitions of the model, which represents our abstract test-case for a particular test goal. In this trace, the state of the system is defined by the set of current locations, the set of data values, and the set of clock values in the TA network. The transitions can be either delay transitions, internal transitions, read, or write transitions.

Assuming the formal model presented in Section 5.1.1 as input and the testing goal, i.e., the functional requirement of the BBW system formalized as a reachability property, the UPPAAL PORT model-checker generates an abstract test-case automatically, a part of which we present in Figure 5.10 (it shows a part of the test showing the path in the ABS to be tested).

```
1 State:(ABSFL.idle)
  ABSFL.w=0 ABSFL.wheelABS=0 ABSFL.torqueABS=-1 ABSFL.v=0 ABSFL.R=1/2
2 Transitions: ABSFL.idle->ABSFL.Entry {w:=8, wheelABS:=1, v:=12}
3 State:(ABSFL.Entry)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1 ABSFL.v=12 ABSFL.R=1/2
4 Transitions: ABSFL.Entry->ABSFL.CalcS {v>0}
5 State:(ABSFL.CalcS)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1 ABSFL.v=12 ABSFL.R=1/2
6 Transitions: ABSFL.CalcS->ABSFL.Exit {v<5*(v-w*R), torqueABS:=0}
7 State: (ABSFL.Exit)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0 ABSFL.v=12 ABSFL.R=1/2
8 Transitions: ABSFL.Exit->ABSFL.idle { }
9 State:(ABSFL.idle)
10 ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0 ABSFL.v=12 ABSFL.R=1/2
```

Figure 5.10: Abstract test-case for the pABS_FL FunctionPrototype.

### 5.4.2 Generation and Execution of Concrete Test-Cases

In order to run the abstract test-cases on the actual implementation, we need to convert them into concrete test-cases, i.e., Python scripts. For this, we automatically parse the abstract test-cases in order to identify the order of states and transitions, together with the values of variables in each state. This information is the basis of our concrete test-case generation, where we create test scripts that send signals with the values of the variables at each state to the target system. The initial values of variables that are sent to the target, in form
of signals, trigger the SUT to execute and evolve through a set of states and transitions. The information about the actual set of states and transitions taken by the SUT during execution is collected and sent back to the script in form of signals. A switch-case structure is used and a dedicated variable keeps track of the current system state, and changes accordingly when moving to a different state. The test script receives the result, the information on the set of states and transitions, as well as the values of variables at each state. This information, collected during execution and at runtime, is then used to compare the actual order of states and transitions against the expected one originated from the models and specified in the abstract test-case. If any discrepancy between the actual and expected orders of traversed states and transitions is found, the test result is evaluated to fail, otherwise a pass verdict is issued. In other words, it is checked that, based on the given inputs, the exact same order of states as in the trace and abstract test-case appears at runtime, during the execution of the system.

We present the concretely Python script generated for the abstract test-case presented in Figure 5.10 in our paper [40] included in the second part of the thesis.

5.5 Our Methodology Overview and Tool Support

In this section we present an overview of the entire formal analysis and model-based testing methodology, and we describe the corresponding tool support.

5.5.1 Methodology Description

Our methodology allows for formal verification [36] and test-case generation starting from EAST-ADL models, down to their execution on the system implementation (SUT) [40]. The methodology is implemented by a tool chain consisting of ViTAL [26, 36, 39] and Farkle [22] (see Figure 5.11), and it contains the following steps:

**Step 1** Create or import the EAST-ADL model in ViTAL.

**Step 2** Select the verification method:

a. Verification with UPPAAL or UPPAAL SMC:
   - Transform the EAST-ADL model into a network of UPPAAL TA.
Chapter 5. Thesis Contribution

- Edit the TA associated with the behavior of the corresponding EAST-ADL FunctionPrototype.
- Analyze the system with the UPPAAL or UPPAAL SMC model-checker (depending on the size of the resulted model).

b. Verification with UPPAAL PORT:
- Create the associated TA behavior to each EAST-ADL FunctionPrototype.
- Transform the EAST-ADL model +TA model into a network of UPPAAL PORT TA.
- Analyze the system with the UPPAAL PORT model-checker.

Step 3 Perform model-based testing starting from the EAST-ADL model:
- Input: the formal model from Step 2b and a testing goal (i.e., a formalized system requirement).
- Generate abstract test-cases with UPPAAL PORT.
- Convert the abstract test-cases into concrete test-cases automatically, by generating Python test scripts executable by Farkle.
- Execute the concrete test-cases against the SUT, to obtain a pass or fail verdict.

5.5.2 ViTAL: a Verification Tool for EAST-ADL Models using UPPAAL PORT

In this thesis, we have developed ViTAL as tool support for our methodology presented in Section 5.5.1. ViTAL integrates architectural languages and verification techniques, to provide simulation, formal verification, and test-case generation for EAST-ADL system models. The tool is based on different Eclipse plug-ins, as depicted in Figure 5.11. The User Interface integrates an editor for EAST-ADL models in the Eclipse framework, as well as an UPPAAL PORT TA editor to model the timing and functional behavior model associated with each EAST-ADL FunctionType. The mapping between these two models is used by the analysis and test-case generation interface of UPPAAL PORT, which uses a client-server architecture.

In a similar manner, a mapping of the EAST-ADL model results in a model that can be opened by the UPPAAL TA editor, and further analyzed in the UPPAAL analysis interface, which uses a client-server architecture. Using these
5.5 Our Methodology Overview and Tool Support

5.5.2 ViTAL: a Verification Tool for EAST-ADL Models using UPPAAL PORT

In this thesis, we have developed ViTAL as tool support for our methodology presented in Section 5.5.1. ViTAL integrates architectural languages and verification techniques, to provide simulation, formal verification, and test-case generation for EAST-ADL system models. The tool is based on different Eclipse plug-ins, as depicted in Figure 5.11. The User Interface integrates an editor for EAST-ADL models in the Eclipse framework, as well as an UPPAAL PORT TA editor to model the timing and functional behavior model associated with each EAST-ADL FunctionType. The mapping between these two models is used by the analysis and test-case generation interface of UPPAAL PORT, which uses a client-server architecture.

In a similar manner, a mapping of the EAST-ADL model results in a model that can be opened by the UPPAAL TA editor, and further analyzed in the UPPAAL analysis interface, which uses a client-server architecture. Using these two verification interfaces, it is possible to validate the behavior and timing of EAST-ADL system models prior to implementation.

For details on Farkle, we refer to reader to the literature [22, 40].

Figure 5.11: Our integrated verification methodology.
5.6 Validation of the Methodology on a Brake-by-Wire Use-Case

Our methodology introduced in Section 5.5, supported by the ViTAL-Farkle tool chain has been applied on the BBW case study for validation purposes. The formal model of the BBW system has been simulated and formally verified against different functional and timing properties with UPPAAL [39], UPPAAL PORT [26, 36], and UPPAAL SMC [39].

We have shown how model-checking techniques can be employed to automatically generate abstract test-cases for functional requirements based on the TA enriched EAST-ADL model of the BBW use-case with UPPAAL PORT. The abstract test-cases are transformed into Python scripts representing the executable test-cases that are finally run on the actual code. Our work is an attempt to exercise the feasibility of test-case generation from EAST-ADL models [40]. The formal verification and model-based testing methodology has shown encouraging results when applied on a Brake-by-Wire prototype (see papers [36, 39, 40]).

5.7 Research Goals Revisited

In this section we present the contribution of each of the four papers and we show how they address the research goals formulated in Section 4.2.
5.7 Research Goals Revisited

5.7.1 Paper A

In this paper [36], we present the use of formal modeling and verification techniques at an early development stage of automotive embedded systems which are originally described in the EAST-ADL architectural language. We employ the UPPAAL PORT timed automata formalism to capture the behavior each Function Prototype in the EAST-ADL model, while the EAST-ADL architectural model presents interactions between components and other related information (Goal 1). The verification framework presented in the paper allows for modeling, simulation, and formal verification of functional and timing requirements of the system based on model-checking techniques (Goal 2). The independence introduced by the read-execute-write semantics, employed by the EAST-ADL functional modeling, is exploited by UPPAAL PORT in order to reduce time and space requirements for model-checking. Our technique improves behavior modeling, verification, and analysis capability of EAST-ADL, and the result shows the applicability of model-checking on automotive complex embedded systems (Goal 5). This framework will be later extended in Paper D to allow for abstract test-case generation based on the EAST-ADL models extended with timed automata semantics (Goal 4).

5.7.2 Paper B

In this paper [39], we present a constellation of complementary verification techniques that can be applied on EAST-ADL models to deliver various types of model correctness assurance. In order to enable the verification of architectural models, we show three verification techniques based on: (i) the simulation of the architectural models with Simulink, (ii) the symbolic simulation and formal verification of the system with UPPAAL (Goal 2) and (iii) the statistical model-checking of the architectural model with UPPAAL SMC (Goal 2). We show how the analysis techniques underlying the tools complement each other, by applying the transformations and tools to analyze the industrial system. The actual contribution of this paper consists in introducing two new transformations, one from EAST-ADL models to Simulink models, and one from EAST-ADL models to UPPAAL models (Goal 1), together with the application of simulation, model-checking and statistical model-checking on an industrial architectural model (Goal 5).
5.7.3 Paper C

In this paper, we provide an overview of the model-based testing approaches for requirements-based models. These approaches are classified based on the model paradigm used to generate the test-cases, and are further detailed based on the test generation method, the test selection criteria, testing artifact and mapping, focusing on the tool support. In order to achieve this, we extend the existing model-based testing taxonomy with the testing artifact and the mapping between abstract and executable test-cases. Our goal is to get a deeper insight into the state-of-the-art in this area as well as form a position with respect to possible needs and gaps in the current tools used by industry and academia, gaps that need to be addressed in order to enhance the applicability of model-based testing (Goal 3).

5.7.4 Paper D

In this paper [40], we introduce a methodology for model-based testing against functional requirements of embedded systems, starting from the EAST-ADL architectural models (Goal 4). As part of the methodology, we show how to generate executable test-cases for the system implementation, starting from abstract tests generated by model-checking EAST-ADL high-level artifacts extended with TA behavior. The main goal of this paper is to check the feasibility of the EAST-ADL+TA generated abstract test-cases by actually running the corresponding executable test-cases on the implemented code, in an attempt to obtain a pass or fail verdict (Goal 5). If the endeavor succeeds, the testing effort could then be reduced by the automatic provision of valid test-cases.
Chapter 6

Related Work

6.1 Analysis of Architectural Models of Embedded Systems

Analyzing software architectures is a challenging task that has received a considerable attention from the research community in the past few years; for instance, Allen [2] applies model-checking techniques directly at the (software) architecture level, whereas Burmester [20] proposes an approach for analyzing safety-critical, real-time systems. Based on such research, Zhang et al. [57] have compared and classified over 16 approaches to analyze software architectures using model-checking techniques, which shows that this research topic is of growing interest and among these methods, and also that testing and formal analysis of such models play an important role. Even though researchers in the software testing community have been investigating how design components and architectural description languages can be used for testing purposes, only a few concrete testing techniques for architectural languages have been proposed [11, 42, 46].

Two architectural languages have received special attention from both academia and industry, EAST-ADL [16] and AADL (Architecture Analysis and Design Language) [27] because of their direct applicability to specific development domains, like the automotive industry. Several works on the analysis of AADLs have been carried out, and an analyzable behavioral and architectural model has been proposed where AADL is enriched with its behavioral annex [9, 15], which is considered as the underlying model used for analysis.
Chapter 6. Related Work

Berthomieu et al. [9] use AADL using the Fiacre language for analyzing the functional behavior with the TINA tool. In addition, Sandberg et al. [48] provide an approach that performs iterative analysis for managing changes in the safety architecture at analysis level to ensure that it still meets function specific safety goals derived at vehicle level. To improve the usage of AADL in practice, Lasnier et al. [37] are proposing an automatic code generation from AADL models and an analysis approach that is applicable to validate specific properties concerning real-time systems. In 2010, Bozzano et al. [17] have proposed a model-checker for AADL focusing on safety, dependability analysis and performance evaluation for AADL models in the aerospace domain. However, none of these frameworks can be directly applied on the EAST-ADL language, nor do they provide a complete model-based testing framework like our proposed analysis framework.

Several methods have been developed for the analysis of EAST-ADL models: Nallet et al. [30] describe the use of UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) together with EAST-ADL for timing analysis [30]. In the context of EAST-ADL UML2 profile, Feng et al. [28] propose a translation of an existing behavioral principle of the system specified in activity diagrams into PROMELA and use the SPIN model-checker for formal verification of EAST-ADL functional models. Qureshi et al. [45] describe an integration effort towards verification of EAST-ADL models based on timing constraints, i.e., the EAST-ADL models are transformed into UPPAAL models that are then validated in terms of the timing- and triggering constraints. Marinescu and Enoiu [38] are describing a method for using resource-usage analysis to provide rigorous integration of resource requirements into EAST-ADL. Compared to the above mention frameworks, our work provides exhaustive verification of both functional and timing properties, and allows the generation of abstract test-case based on model-checking techniques.

6.2 Testing Architectural Models starting from Requirements Specification

Testing of software using architectural languages has been proposed by fewer researchers, starting with Bertolino et al. in 1996 [11]. Muccini and Bertolino have studied further this subject by performing different case studies that show the insights gain and some issues in architecture-based software testing [10, 42, 12]. Specifically, their research shows how to identify suitable tests for complex real-world systems. This topic has recently raised some interest, from
6.2 Testing Architectural Models starting from Requirements Specification

defining architectural-based testing criteria [47], and using software architecture for regression testing [33] to automated generation of system level tests [1]. Testing software architecture languages has been used to check the considered model itself, to test the system implementation, or its conformance to the requirements expressed by the software architecture.

The combination of analysis and testing of architectural languages was been loosely considered until now, and to the best of our knowledge there is no practical solution that can be generally applied and used in an industrial setting. For more details, we refer the reader to the current work by Bertolino [13], in which the author has reviewed the research in software architecture-based analysis and testing. In this context, our work provides an integrated framework for formal analysis and test-case generation of EAST-ADL architectural description, based on model checking techniques. The framework is supported by the ViTAL tool, and the interconnection with the Farkle tool provides support for the execution of the generated test-cases.
In this thesis, we have proposed a methodology for the analysis of, and test-case generation starting from, EAST-ADL architectural models that are used by various automotive companies. The methodology provides behavior for EAST-ADL models as timed automata networks in order to make possible the formal analysis of such models based on different model-checking techniques. In our work, we employ on: (i) UPAPAAL, a well-known model-checker for real-time systems, (ii) UPAPAAL PORT, an extension of the UPAPAAL model-checker for component-based systems, and (iii) UPAPAAL SMC, an extension of UPAPAAL for statistical model-checking. The UPAPAAL PORT model-checker is based on partial order reduction techniques, and the analysis is performed on a symbolic representation of the hierarchical component structure of the input model without the usual transformation to a “flat” network of timed automata. Unlike symbolic model-checking, statistical model-checking can show results only up to a specified level of confidence and never for certain like symbolic techniques, yet it is a cheap way to generate and confirm safety counter-examples where symbolic techniques may employ expensive over-approximations. Our analysis methodology relies on formal verification of the original EAST-ADL model by model-checking based on above mentioned techniques, where the symbolic and statistical techniques complement each other.

In order to collect information with respect to possible needs and gaps of the current model-based testing methods used by industry and academia, we have overviewed the state-of-the-art in the area of requirements-driven model-
Chapter 7

Conclusions and Future Work

In this thesis, we have proposed a methodology for the analysis of, and test-case generation starting from, EAST-ADL architectural models that are used by various automotive companies. The methodology provides behavior for EAST-ADL models as timed automata networks in order to make possible the formal analysis of such models based on different model-checking techniques. In our work, we employ on: (i) UPAAAL, a well known model-checker for real-time systems, (ii) UPAAAL PORT, an extension of the UPAAAL model-checker for component-based systems, and (iii) UPAAAL SMC, an extension of UPAAAL for statistical model-checking. The UPAAAL PORT model-checker is based on partial order reduction techniques, and the analysis is performed on a symbolic representation of the hierarchical component structure of the input model without the usual transformation to a “flat” network of timed automata. Unlike symbolic model-checking, statistical model-checking can show results only up to a specified level of confidence and never for certain like symbolic techniques, yet it is a cheap way to generate and confirm safety counter-examples where symbolic techniques may employ expensive over-approximations. Our analysis methodology relies on formal verification of the original EAST-ADL model by model-checking based on above mentioned techniques, where the symbolic and statistical techniques complement each other.

In order to collect information with respect to possible needs and gaps of the current model-based testing methods used by industry and academia, we have overviewed the state-of-the-art in the area of requirements-driven model-
based testing. The result of the literature review has motivated us to extend our formal analysis methodology of EAST-ADL model with model-based testing features that allow us to automatically generate and execute test-cases for the system implementation. We have focused on generating abstract test-cases for functional requirements, and we have used the UPPAAL PORT model-checker’s ability to generate witness traces for reachability properties, from the EAST-ADL model extended with UPPAAL PORT TA semantics. The resulting abstract test-cases are transformed into Python scripts that represent the concrete test-cases that are finally run on the implementation code to get pass or fail verdicts, with respect to the tested requirement.

The work presented in this thesis is an attempt to check the feasibility of analyzing architectural models (i.e, EAST-ADL models) by model-checking techniques, and it has been implemented in a tool called ViTAL. ViTAL integrates model-checking techniques with EAST-ADL models to provide formal analysis and generation of abstract test-cases. The ViTAL tool has been paired with the Farkle tool, to enable automatic abstract test-case conversion and execution on the system implementation. The entire methodology has been applied on the Brake-by-Wire case-study and has shown encouraging results.

As future work, we plan to extend our methodology for test-case generation against timing requirements, such as end-to-end deadlines. We will continue our research by investigating how to use timed abstract test-cases generated from EAST-ADL + TA models for testing the implementation (obtained based on the model). Furthermore, we intend to investigate how architectural languages, such as EAST-ADL, can be used to guide the generation of test-case. To achieve this, we will provide a set of architectural coverage criteria, which will ensure a true conformance testing between the EAST-ADL model and the system implementation. In addition, we intend to verify the applicability of our framework on other architectural or component-based systems.
Chapter 7. Conclusions and Future Work

Based testing. The result of the literature review has motivated us to extend our formal analysis methodology of EAST-ADL model with model-based testing features that allow us to automatically generate and execute test-cases for the system implementation. We have focused on generating abstract test-cases for functional requirements, and we have used the UPPAAL PORT model-checker’s ability to generate witness traces for reachability properties, from the EAST-ADL model extended with UPPAAL PORT TA semantics. The resulting abstract test-cases are transformed into Python scripts that represent the concrete test-cases that are finally run on the implementation code to get pass or fail verdicts, with respect to the tested requirement.

The work presented in this thesis is an attempt to check the feasibility of analyzing architectural models (i.e., EAST-ADL models) by model-checking techniques, and it has been implemented in a tool called ViTAL. ViTAL integrates model-checking techniques with EAST-ADL models to provide formal analysis and generation of abstract test-cases. The ViTAL tool has been paired with the Farkle tool, to enable automatic abstract test-case conversion and execution on the system implementation. The entire methodology has been applied on the Brake-by-Wire case-study and has shown encouraging results.

As future work, we plan to extend our methodology for test-case generation against timing requirements, such as end-to-end deadlines. We will continue our research by investigating how to use timed abstract test-cases generated from EAST-ADL + TA models for testing the implementation (obtained based on the model). Furthermore, we intend to investigate how architectural languages, such as EAST-ADL, can be used to guide the generation of test-case. To achieve this, we will provide a set of architectural coverage criteria, which will ensure a true conformance testing between the EAST-ADL model and the system implementation. In addition, we intend to verify the applicability of our framework on other architectural or component-based systems.

Bibliography


in software development (Viewpoints’ 96) on SIGSOFT’96 workshops, pages 62–64. ACM, 1996.


[47] D. J. Richardson and A. L. Wolf. Software testing at the architectural level. In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints’ 96) on SIGSOFT’96 workshops, pages 68–71. ACM, 1996.


Chapter 7. Conclusions and Future Work


