MODEL-DRIVEN ANALYSIS AND VERIFICATION OF AUTOMOTIVE EMBEDDED SYSTEMS

Raluca Marinescu

2016

School of Innovation, Design and Engineering
MODEL-DRIVEN ANALYSIS AND VERIFICATION
OF AUTOMOTIVE EMBEDDED SYSTEMS

Raluca Marinescu

Akademin för innovation, design och teknik

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 7 oktober 2016, 13.15 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor David Garlan, Carnegie Mellon University
Abstract

Modern vehicles are equipped with electrical and electronic systems that implement highly complex functions, such as anti-lock braking, cruise control, etc. To realize and integrate such complex embedded systems, the automotive development process requires an updated methodology that takes into consideration the system’s intricate features and examines both their functional and extra-functional requirements. Early design artifacts like architectural models represent convenient abstractions for reasoning about the system’s structure and functionality. In this context, the EAST-ADL language has been developed as a domain-specific architectural language that targets the automotive industry and is aligned with the AUTOSAR automotive standard. To fully enjoy the benefits of these abstract system descriptions, architectural models need to be integrated into a model-driven development framework that enables also verification by, e.g., model checking and model-based testing. One major drawback in developing such a framework lies in the fact that architectural models, while capturing the system’s structure and inter-component communication, often lack direct means to represent the desired internal behavior of the system in a semantically well-defined way. To overcome this, one needs to provide means of integrating both structural as well as behavioral information, desirably within the same framework backed by formal semantics, in order to enable the model’s formal verification.

In this thesis, we propose a tool-supported integrated formal modeling and verification framework tailored for automotive embedded systems that are originally described in the EAST-ADL architectural language. To achieve this, we first provide formal semantics to the architectural model and its behavior by proposing an equivalent formal description as a network of timed automata. This enables us to analyze the resulting network of timed automata formally by model checking, using both the UPPAAL PORT and UPPAAL SMC model checkers. UPPAAL PORT is providing efficient component-aware verification via the partial order reduction technique, while UPPAAL SMC is extending UPPAAL with statistical model-checking capabilities via probabilistic algorithms. We focus the analysis on functional and timing requirements, but also on the system’s resource usage with respect to different resources specified in the model, such as memory and energy. In an attempt to narrow the gap between the original architectural model and the eventual system implementation, we define an executable semantics of the UPPAAL PORT components that guarantees that the implementation preserves the invariant properties of the model. Assuming a system implementation that conforms to the formal model, we investigate how to provide test cases suitable for the eventual verification of such implementation, by exploiting the model checker’s ability to generate witness traces for reachability verification. Such a witness trace represents a execution of the system from its initial state to the goal state encoded by the reachability property, and becomes our abstract test case. 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 on the system under test, which ends up in pass/fail testing verdicts. Dependency analysis is a method that is able to identify crucial intra- and inter-component dependencies early in the system’s development life cycle, if applied on architectural models. In this thesis, we also investigate how such dependencies, resulting from applying dependency analysis on EAST-ADL models, can be exploited during formal verification in order to reduce the verified state-spaces during model checking. The framework is supported by the ViTAL tool and its applicability is shown on an automotive industrial prototype, namely a Brake-by-Wire system.
Abstract

Modern vehicles are equipped with electrical and electronic systems that implement highly complex functions, such as anti-lock braking, cruise control, etc. To realize and integrate such complex embedded systems, the automotive development process requires an updated methodology that takes into consideration the system’s intricate features and examines both their functional and extra-functional requirements. Early design artifacts like architectural models represent convenient abstractions for reasoning about the system’s structure and functionality. In this context, the EAST-ADL language has been developed as a domain-specific architectural language that targets the automotive industry and is aligned with the AUTOSAR automotive standard. To fully enjoy the benefits of these abstract system descriptions, architectural models need to be integrated into a model-driven development framework that enables also verification by, e.g., model checking and model-based testing. One major drawback in developing such a framework lies in the fact that architectural models, while capturing the system’s structure and inter-component communication, often lack direct means to represent the desired internal behavior of the system in a semantically well-defined way. To overcome this, one needs to provide means of integrating both structural as well as behavioral information, desirably within the same framework backed by formal semantics, in order to enable the model’s formal verification.

In this thesis, we propose a tool-supported integrated formal modeling and verification framework tailored for automotive embedded systems that are originally described in the EAST-ADL architectural language. To achieve this, we first provide formal semantics to the architectural model and its behavior by proposing an equivalent formal description as a network of timed automata. This enables us to analyze the resulting network of timed automata formally by model checking, using both UPPAAL PORT and UPPAAL SMC, two extensions of the UPPAAL model checker. UPPAAL PORT is providing efficient component-aware verification via the partial order reduction technique, while UPPAAL SMC is extending UPPAAL with statistical model-checking capabilities via probabilistic algorithms. We focus the analysis on functional and timing requirements, but also on the system’s resource usage with respect to different resources.
Specifikt i moderna fordon finns inbyggda elektroniska system som styr komplexa funktioner, såsom ABS-bromsar, farthallare, mm. För att kunna realisera och integrera så komplexa system i fordon krävs att fordonsindustrins utvecklingsprocesser och metoder tar hänsyn till systemens komplicerade funktionella och extrafunktionella egenskaper och krav. Tidiga designartefakter som till exempel arkitekturmodeller utgör en passande abstraktion för att kunna representera systemens struktur och funktionalitet. För detta ändamål har spraket EAST-ADL tagits fram som ett domän specifierat arkitekturspråk för fordonsindustrin. För att fullfölja kunna utnyttja fördelarna med använda beskrivningar av dessa abstrakta representationer, så krävs att arkitekturmodellerna kan integreras med ett modelldrivet utvecklingsramverk som tillåter verifiering genom till exempel model-checking eller modellbaserad testning. En av de största utmaningarna med att utveckla ett sådant ramverk är att arkitekturmodeller ger en bra beskrivning av systemets struktur, men ofta saknar en väldefinierad semantisk representation för att beskriva delsystemens interna beteenden. För att kunna formellt verifiera en modell så krävs att både struktur och beteende finns beskrivna, helst inom ett och samma ramverk, samt att detta ramverk har en väldefinierad formell semantik.

I denna avhandling presenteras ett verktygsett ramverk för formell modellering och verifiering. Ramverket är skrattatsytt för inbyggda system inom fordonsindustrin och arkitekturspråket EAST-ADL. Vi föreslår en formell semantik baserad på network of timed automata (en) vilken fullständigt beskriver beteendet i en arkitekturmodell. Detta ger oss möjlighet att genomföra formell analys med hjälp av verktygen UPPAAL PORT och UPPAAL SMC. UPPAAL PORT utför effektiv komponentmedveten verifiering genom så kallad partial-order reduction (en), medan UPPAAL SMC utökar UPPAAL med så kallad statistical model-checking (en) genom probabilistiska algoritmer. Vi fokuserar på analys och verifiering av funktionella krav och tidskrav, men också systems resursanvändning, där de olika resurserna kan specificeras i modellen som till exempel minnes- eller energianvändning. I ett försök att minska skillnaden mellan arkitekturmodellen och den slutgiltiga systemimplementeringen, definierar vi en exekverbar semantik av UPPAAL PORT-komponenter som garanterar att implementeringen bevarar de invarianta egenskaperna i modellen. Under antagandet att ett system implementering som konform med formal modell, experimenterar vi hur vi kan tillhandahålla testfall passande för den eventuella verifieringen av sådan implementering, genom exploitation av model checkers ability to generate witness traces for reachability verification. Such witness traces represent executions of the system from its initial state to the goal state encoded by the reachability property, and becomes our abstract test case. 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 on the system under test, which ends up in pass/fail testing verdicts. Dependency analysis is a method that is able to identify crucial intra- and inter-component dependencies early in the system’s development life cycle, if applied on architectural models. In this thesis, we also investigate how such dependencies, resulting from applying dependency analysis on EAST-ADL models, can be exploited during formal verification in order to reduce the verified state-spaces during model checking. The framework is supported by the ViTAL tool and its applicability is shown on an automotive industrial prototype, namely a Brake-by-Wire system.
Sammanfattning


I denna avhandling presenteras ett verktygsstött ramverk för formell modellering och verifiering. Ramverket är skråddarsytt för inbyggda system inom fordon industri och arkitekturspråket EAST-ADL. Vi föreslår en formell semantik baserad på network of timed automata (eng) vilken fullständigt beskriver beteendet i en arkitekturmodell. Detta ger oss möjlighet att genomföra formell analys med hjälp av verktygen UPPAAL PORT och UPPAAL SMC. UPPAAL PORT utför effektiv komponentmedveten verifiering genom så kallad partial-order reduction (eng), medan UPPAAL SMC utökar UPPAAL med så kallad statistical model-checking (eng) genom probabilistiska algoritmer. Vi fokuserar på analys och verifiering av funktionella krav och tidskrav, men också på systems resursanvändning, där de olika resurserna kan speciferas i modellen som till exempel minnes- eller energianvändning. I ett försök att minska skillnaden mellan arkitekturmodellen och den slutgiltiga systemimplementeringen, definierar vi en exekverbar semantik av UPPAAL PORT-komponenter som garanterar att implementationen bevarar de invarianter egenskaperna i modellen. Under antagandet att ett sys-
tems implementation och formella modell överensstämmer, så har vi undersökt hur man kan ta fram testfall som passar för verifiering av en sådan implementation, genom att använda oss av möjligheten i model-checking att generera så kallade witness traces vid vissa verifieringar. En sådan witness trace representerar en körning av ett system från dess initiala tillstånd till ett måltillstånd, vilket då kommer att utgöra ett abstrakt testfall. Genom att para samman den automatiserade modellbaserade testfallsgeneratorken med en automatiserad transformation från de abstrakta testfallen till Python-skript, så kan vi exekvera de genererade Python-skripten på system som ska testas, och därigenom nå ett testutfall.

Dependability analysis är en metod för att identifiera viktiga interna och externa komponentberoenden tidigt i systemutvecklingsprocessen, givet att den appliceras på en arkitekturmodell. Vi har i den här avhandlingen undersökt hur beroenden som kommer från en dependability analysis av EAST-ADL-modeller kan användas i formell verifiering för att minska antalet tillstånd som behöver verifieras med hjälp av model-checking. Ramverket stöds av verktyget ViTAL och dess tillämplighet i fordonsindustrin demonstreras på ett “Break-by-Wire” system.
items implementation och formella modell överensstämmer, så har vi undersökt hur man kan ta fram testfall som passar för verifiering av en sådan implementation, genom att använda oss av möjligheten i model-checking att generera så kallade witness traces vid vissa verifieringar. En sådan witness trace representerar en körning av ett system från dess initiala tillstånd till ett måltillstånd, vilket då kommer att utgöra ett abstrakt testfall. Genom att para samman den automatiserade modellbaserade testfallsgeneratorn med en automatiserad transformation från de abstrakta testfallen till Python-skript, så kan vi exekvera de genererade Python-skripten på system som ska testas, och därgentom nå ett testutfall.

Dependability analysis är en metod för att identifiera viktiga interna och externa komponentberoenden tidigt i systemutvecklingsprocessen, givet att den appliceras på en arkitekturmodell. Vi har i den här avhandlingen undersökt hur beroenden som kommer från en dependability analysis av E AST-ADL-modeller kan användas i formell verifiering för att minska antalet tillstånd som behöver verifieras med hjälp av model-checking. Ramverket stöds av verktyget ViTAL och dess tillämpighet i fordonsindustrin demonstreras på ett "Break-by-Wire" system.

"Discovery consists of seeing what everybody has seen and thinking what nobody else has thought."
– Albert Szent-Györgyi
Acknowledgments

I would like to take this opportunity to thank the people without whom this thesis would have never been written.

First and foremost, I would like to thank my two amazing supervisors, Associate Professor Cristina Seceleanu and Professor Paul Pettersson. Your passion for research, your patience, your positive and energetic attitude, have been truly inspiring. Thank you for giving me the opportunity to become a PhD student, and thank you for guiding and supporting me during this long and beautiful journey.

Secondly, I would like to thank the many researchers with whom I had the great pleasure to collaborate and co-author papers during my PhD studies: Eun-Young Kang, Eduard Paul Enoiu, Pierre-Yves Schobbens, Henrik Kaijser, Marius Mikuˇcionis, Henrik L ¨onn, Alexandre David, H `el´ene Le Guen, Mehrdad Saadatmand, Andreas Hammar, Detlef Scholle, Elaine Weyuker, Saad Mubeen, Predrag Filipovikj, Nesredin Mahmud, Alessio Bucaioni, Oscar Ljungkrantz and Aida Causevic. I am also grateful to all the wonderful researchers at M¨alardalen University, both senior researchers and fellow PhD students, for all the wonderful moments we have shared together during lectures, research meetings, and coffee breaks.

Many thanks to my family for their love and support through all my years of university studies. I would also like to thank my husband, Eduard, for believing in me and being there for me through the best and the hardest of times. Thank you for your enthusiasm and support, and thank you for the great ideas and the time you have invested in our joint papers.

Last but not least, I would like to thank the committee members, Professor Antonia Bertolino, Associate Professor Bernhard Aichernig, and Associate Professor Brian Nielsen, who have kindly agreed to review and grade my thesis. I am also extremely grateful to Professor David Garlan for accepting to be my faculty examiner. Thank you for your time, generosity, and effort.

Raluca Marinescu

V¨aster˚as, Sweden

August 26, 2016
Acknowledgments

I would like to take this opportunity to thank the people without whom this thesis would have never been written.

First and foremost, I would like to thank my two amazing supervisors, Associate Professor Cristina Secleeanu and Professor Paul Pettersson. Your passion for research, your patience, your positive and energetic attitude, have been truly inspiring. Thank you for giving me the opportunity to become a PhD student, and thank you for guiding and supporting me during this long and beautiful journey.

Secondly, I would like to thank the many researchers with whom I had the great pleasure to collaborate and co-author papers during my PhD studies: Eun-Young Kang, Eduard Paul Enoiu, Pierre-Yves Schobbens, Henrik Kajser, Marius Mikučionis, Henrik Lönn, Alexandre David, Hélène Le Guen, Mehrdad Saadatmand, Andreas Hammar, Detlef Scholle, Elaine Weyuker, Saad Mubeen, Predrag Filipovikj, Nesredin Mahmud, Alessio Bucaioni, Oscar Ljungkrantz and Aida Causevic. I am also grateful to all the wonderful researchers at Mälardalen University, both senior researchers and fellow PhD students, for all the wonderful moments we have shared together during lectures, research meetings, and coffee breaks.

Many thanks to my family for their love and support through all my years of university studies. I would also like to thank my husband, Eduard, for believing in me and being there for me through the best and the hardest of times. Thank you for your enthusiasm and support, and thank you for the great ideas and the time you have invested in our joint papers.

Last but not least, I would like to thank the committee members, Professor Antonia Bertolino, Associate Professor Bernhard Aichernig, and Associate Professor Brian Nielsen, who have kindly agreed to review and grade my thesis. I am also extremely grateful to Professor David Garlan for accepting to be my faculty examiner. Thank you for your time, generosity, and effort.

Raluca Marinescu
Västerås, Sweden
August 26, 2016
List of Publications

Papers Included in the Doctoral Thesis

Paper A:
A Methodology for Formal Analysis and Verification of EAST-ADL models

Paper B:
Analyzing Industrial Architectural Models by Simulation and Model Checking

Paper C:
Statistical Analysis of Resource Usage of Embedded Systems Modeled in EAST-ADL

Paper D:
A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs

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

Papers Included in the Doctoral Thesis

**Paper A: A Methodology for Formal Analysis and Verification of EAST-ADL models**

**Paper B: Analyzing Industrial Architectural Models by Simulation and Model Checking**

**Paper C: Statistical Analysis of Resource Usage of Embedded Systems Modeled in EAST-ADL**

**Paper D: A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs**

---

1The included papers have been reformatted to comply with the thesis layout.
2Reprinted with permission.


Other Peer-reviewed Publications


Contents

I Thesis 1

1 Introduction 3

1.1 Thesis Overview......................... 8

2 Preliminaries 15

2.1 Model-driven Development . . . . . . . . . . . . . . . . . . 15

2.2 Architectural Modeling: E AST-ADL .............. 16

2.2.1 Industrial Use Case: The Brake-by-Wire System . . . 19

2.3 Methods for System Analysis and Verification . . . . . . . . . 22

2.3.1 Symbolic Simulation . . . . . . . . . . . . . . . . . . 23

2.3.2 Model Checking .................... 23

2.3.3 Statistical Model Checking . . . . . . . . . . . . . . . 28

2.3.4 Model-based Testing . . . . . . . . . . . . . . . . . . 31

2.3.5 Dependency Analysis of Architectural Models . . . . 32

3 Research Focus 35

3.1 Problem Description ...................... 35

3.2 Research Goals ......................... 36

4 Research Methodology 39

5 Thesis Contributions 43

5.1 Overview of the Proposed Framework . . . . . . . . . . . . . 43

5.2 Formal Semantics of E AST-ADL Models............ 44

5.3 Formal Analysis of E AST-ADL Models ............. 48

5.4 Tool supported Model-based Testing: Literature Review . . . 49

5.5 Model-based Testing of Automotive Systems . . . . . . . . 50
# Contents

## I  Thesis  

1  Introduction  
   1.1  Thesis overview  

2  Preliminaries  
   2.1  Model-driven Development  
   2.2  Architectural Modeling: EAST-ADL  
      2.2.1  Industrial Use Case: The Brake-by-Wire System  
   2.3  Methods for System Analysis and Verification  
      2.3.1  Symbolic Simulation  
      2.3.2  Model Checking  
      2.3.3  Statistical Model Checking  
      2.3.4  Model-based Testing  
      2.3.5  Dependency Analysis of Architectural Models  

3  Research Focus  
   3.1  Problem Description  
   3.2  Research Goals  

4  Research Methodology  

5  Thesis Contributions  
   5.1  Overview of the Proposed Framework  
   5.2  Formal Semantics of EAST-ADL Models  
   5.3  Formal Analysis of EAST-ADL Models  
   5.4  Tool supported Model-based Testing: Literature Review  
   5.5  Model-based Testing of Automotive Systems  

xiii
8.6.2 Analysis and Verification ........................................... 118
8.7 Related Work ......................................................... 120
8.8 Conclusion and Future Work ......................................... 121
Bibliography ................................................................. 122

9 Paper B:
Analyzing Industrial Architectural Models by Simulation and Model
Checking ................................................................. 127
9.1 Introduction ............................................................ 129
9.2 Brief Overview of the EAST-ADL Language ....................... 130
9.3 The Current Development Process in an Automotive Context . 131
9.4 Our Methodology for Analyzing Architectural Models ........... 132
9.5 An Example from Industry: Brake-by-Wire Case Study ............ 134
9.6 Simulation of EAST-ADL Functional Architecture in Simulink 136
9.7 Formal Semantics of EAST-ADL as a network of Timed Automata 140
9.8 Analysis of EAST-ADL Models Using Model
    Checking and Statistical Model Checking .......................... 142
9.9 Related Work .......................................................... 145
9.10 Conclusions and Discussion ........................................... 146
Bibliography ................................................................. 147

10 Paper C:
Statistical Analysis of Resource Usage of Embedded Systems Modeled in EAST-ADL 151
10.1 Introduction ............................................................ 153
10.2 Preliminaries .......................................................... 154
    10.2.1 EAST-ADL Language ........................................... 154
    10.2.2 The ViTAL tool ................................................ 155
    10.2.3 UPPAAL SMC ................................................. 156
10.3 Methodology Overview of Resource Analysis ....................... 156
10.4 Analyzing Resource-usage of EAST-ADL Functions .............. 158
    10.4.1 The Priced Timed Automata Model ........................... 158
    10.4.2 Resource Analysis ............................................. 159
10.5 Applying SMC on the Brake-by-Wire System ....................... 160
    10.5.1 Brake-by-Wire System ....................................... 160
    10.5.2 Applying ViTAL on the Brake-By-Wire System ............ 161
    10.5.3 The Monitor .................................................. 162
    10.5.4 Analysis of Energy Consumption ......................... 163
    10.5.5 Analysis of Memory Usage ................................... 164
11 Paper D:  
A Research Overview of Tool-Supported Model-based Testing of  
Requirements-based Designs  

11.1 Introduction ........................................... 173  
11.2 The Generic Model-based Testing Approach ............. 175  
11.3 Proposed Taxonomy Dimensions ........................ 177  
   11.3.1 The modeling notation ............................. 178  
   11.3.2 The test artifact ................................... 179  
   11.3.3 Test selection criteria .............................. 180  
   11.3.4 The test generation method ........................ 181  
   11.3.5 The technology .................................... 182  
   11.3.6 The mapping ....................................... 183  
11.4 A Research Review of Model-based Testing Tools ......... 183  
   11.4.1 Selection criteria and procedures for including/exclud-  
       ing model-based testing tools ......................... 184  
   11.4.2 Our Taxonomy ..................................... 187  
11.5 Running Example: The Coffee/Tea Vending Machine ....... 187  
11.6 Model-based Testing Tools for Pre/Post Notations ...... 188  
   11.6.1 The Z language ................................... 189  
   11.6.2 The B-method .................................... 189  
   11.6.3 Spec# ........................................... 191  
   11.6.4 AsmL ............................................ 192  
   11.6.5 The Coffee/Tea Vending Machine in ProTest........ 192  
11.7 Model-based Testing Tools for Transition-based Notations 194  
   11.7.1 Finite State Machines .............................. 194  
   11.7.2 Labeled Transition Systems ......................... 197  
   11.7.3 Timed Automata .................................. 199  
   11.7.4 UML statecharts .................................. 200  
   11.7.5 The Coffee/Tea Vending Machine in UPPAAL CoVer .. 202  
11.8 Model-based Testing Tools for Stochastic Models ........ 204  
   11.8.1 Markov Chains ................................... 205  
   11.8.2 The Coffee/Tea Vending Machine in MaTeLo ........ 209  
11.9 Model-based Testing Tools for Data-Flow Models ........ 209  
   11.9.1 Simulink, Lustre and Function Block Diagram ....... 209
11.9.2 The Coffee/Tea Vending Machine in CompleteTest. 211
11.10 Results and Discussion 211
11.11 Conclusions 216
Bibliography 217

12 Paper E:
Testing Automotive Embedded Systems Against Functional and Timing Requirements: From EAST-ADL to Code 227
12.1 Introduction 229
12.2 Preliminaries 231
  12.2.1 Model-based Testing 231
  12.2.2 EAST-ADL 232
  12.2.3 ViTAL 233
  12.2.4 Extended Farkle 235
12.3 The Brake-By-Wire Use-case 236
12.4 Testing Method: From EAST-ADL to Code 239
  12.4.1 Framework Overview 240
  12.4.2 Modeling Activities 241
  12.4.3 Implementation Activities 242
  12.4.4 Testing Activities 245
12.5 Method applied on the BBW 250
  12.5.1 Creating the formal model 250
  12.5.2 Code implementation 251
  12.5.3 Testing for functional and timing properties 252
  12.5.4 Testing timing properties at integration level 256
12.6 Reflections on our method’s relevance in practice 258
  12.6.1 Testing activities in practice 260
  12.6.2 Discussion with respect to our method 261
12.7 State of the Art 261
12.8 Conclusions and future work 264
Bibliography 264

13 Paper F:
Pruning Architectural Models of Automotive Embedded Systems via Dependency Analysis 269
13.1 Introduction 271
13.2 Preliminaries 272
  13.2.1 EAST-ADL 273
  13.2.2 UPPAAL 273
13.2.3 ViTAL ........................................... 275
13.2.4 Analytical Analysis of End-to-end Delays in Architectural Models .................................. 276
13.3 Model Pruning Based on Dependency Analysis .......... 278
  13.3.1 Dependency Matrix Generation .......................... 280
  13.3.2 Parsing the Dependency Matrix .......................... 282
  13.3.3 Pruning the EAST-ADL Model .......................... 283
  13.3.4 Safety Property Preservation by Pruned TA Models ... 284
13.4 The Brake-by-Wire Use-case ................................. 285
  13.4.1 Formal Modeling and Requirements Specification .... 287
  13.4.2 Generating the Dependency Matrix ....................... 288
  13.4.3 Pruning based on End-to-end Deadline Requirements .. 289
  13.4.4 A Formal Approach to Compute End-to-end Delays .... 291
  13.4.5 An Analytical Approach to Calculate End-to-end Delays 291
13.5 Discussions ............................................ 292
13.6 Related Work ............................................ 293
13.7 Conclusions and Future Work ............................... 294
Bibliography ............................................... 294
Chapter 1
Introduction

In order to implement complex functions like cruise control and automatic braking, the automotive industry has gone through a technology shift: older technologies, such as mechanical or hydraulic systems, have been partially replaced by electric and electronic components. This has enabled software applications to take a quintessential role in the development of new automotive features. Initially, the software-based solutions were local, isolated and unrelated (e.g., controlling the ignition). Nowadays, modern cars contain close to 100 million lines of software code, which is executed on 70 to 100 microprocessor-based electronic control units (ECUs) connected through buses like CAN, FlexRay, etc. This complexity raises reliability issues, since the code needs to obey tight safety critical requirements, such as functional and timing requirements, resource consumption, etc.

In this context, such complex systems could benefit from a systematic top-down design approach, and to achieve it, one needs to consider the heterogeneity of requirements, component characteristics, etc. Consequently, the automotive industry is moving towards a model-based development and verification process that can handle the associated complexity, while providing early insights into the system's behavior. Using the Simulink tool has already become state of practice in the automotive industry, as it is equipped with modeling, simulation, and code generation capabilities. Simulink models are also used for improving the understanding of the system and of the requirements, as well as to facilitate communication with stakeholders and programmers. Another appealing solution is the use of architecture description languages.
Chapter 1

Introduction

In order to implement complex functions like cruise control and automatic braking, the automotive industry has gone through a technology shift: older technologies, such as mechanical or hydraulic systems, have been partially replaced by electric and electronic components. This has enabled software applications to take a quintessential role in the development of new automotive features. Initially, the software-based solutions were local, isolated and unrelated (e.g., controlling the ignition). Nowadays, modern cars contain close to 100 million lines of software code, which is executed on 70 to 100 microprocessor-based electronic control units (ECUs)\(^1\) connected through buses like CAN, FlexRay, etc. This complexity raises reliability issues, since the code needs to obey tight safety critical requirements, such as functional and timing requirements, resource consumption, etc.

In this context, such complex systems could benefit from a systematic top-down design approach, and to achieve it, one needs to consider the heterogeneity of requirements, component characteristics, etc. Consequently, the automotive industry is moving towards a model-based development and verification process that can handle the associated complexity, while providing early insights into the system’s behavior. Using the Simulink tool [31] has already become state of practice in the automotive industry, as it is equipped with modeling, simulation, and code generation capabilities. Simulink models are also used for improving the understanding of the system and of the requirements, as well as to facilitate communication with stakeholders and programmers.

Another appealing solution is the use of architecture description languages

\(^{1}\)Numbers taken from “This Car Runs on Code”, IEEE Spectrum, 2009.
(ADLs) that can be introduced earlier in the development process, to provide the system’s structure as a set of interacting components. At this abstraction level, the first design decisions are made and the properties of the components are defined, which include the services provided, the performance characteristics, and even the resource usage of components. The architectural model specifies both the structure and the functionality of the system: what the system should do, what are the relations within and between components, how the components should interact, cooperate and synchronize. The model can also capture related information such as timing properties and other extra-functional requirements (e.g., resource-usage constraints), as well as component triggering annotations. Such decisions, made at the architectural level, impact the final implementation of the system with respect to its correctness, performance, predictability, etc. This, in turn, means that the architectural model should be analyzed to provide early, valuable insight with respect to the system’s behavior. Several verification frameworks have already been proposed for different architectural languages, such as AADL [32, 25, 106, 34] or ACME [109, 62, 122].

EAST-ADL [33] is an ADL dedicated to the development of automotive embedded systems and aligned with the AUTOSAR (AUTomotive Open System ARchitecture) standard [1, 56]. In EAST-ADL, the definition of a system is given at different levels of abstraction, representing different stages in the development process. At each abstraction level, the language represents the automotive electrical and electronic systems with sufficient detail to allow modeling for documentation, design, analysis, and synthesis. To enjoy the fully-fledged advantages of reasoning, the EAST-ADL language would benefit from a verification framework that provides, ideally, both formal verification and model-based testing capabilities. This adds means to ensure that both the architectural model and its implementation conform with the system requirements. This is also encouraged by the emergence of new safety standards for the automotive industry, such as ISO 26262 "Road Vehicles - Functional Safety"\(^2\), which specifies that the development process must provide evidence that the requirements are satisfied at each level of abstraction and that the requirements should be traceable across the different levels.

However, addressing such concerns is not an easy task since EAST-ADL, like other ADLs, lacks support to formally specify and analyze the internal behavior of the components. The latter is usually described outside EAST-ADL in semi-formal languages such as UML [104] or Simulink [93]. This also lim-
its the analysis with tools associated with such environments, like the UML tools or the Simulink Design Verifier, respectively. Moreover, such tools do not enable exhaustive analysis for timing constraints, or quantitative analysis of resource usage, which are provided by dedicated model checkers (e.g., UPPAAL [22]) instead.

To alleviate this problem, substantial research effort has been dedicated to transform or integrate verification frameworks with EAST-ADL. Nallet et al. [65] propose the use of UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) for timing analysis of EAST-ADL models, while Feng et al. [55] propose a translation of an EAST-ADL system model to a PROMELA activity diagrams and use the SPIN model checker for formal verification of EAST-ADL functional models. Qureshi et al. [102] describe an integration effort towards verification of EAST-ADL models based on timing- and triggering constraints, where the EAST-ADL models are transformed into UPPAAL [17] models. However, none of these works provides exhaustive verification of both functional and extra-functional properties, and none provides methods of testing EAST-ADL models or their implementations.

These findings have kindled our motivation to introduce a methodology for the formal analysis and model-based testing of automotive embedded systems, starting from their EAST-ADL architectural specifications. To achieve our goal, we first need to define formal semantics for the EAST-ADL architectural language [73]. Due to the fact that automotive systems’ requirements are a blend of functional and timing constraints, we propose to use timed automata (TA) [12] for this task. The formalism allows modeling of functional and timing constraints in a dense-time semantics, and it is also backed by automated verification support, namely the UPPAAL model checker. UPPAAL PORT [68] is a specialized extension of the UPPAAL model checker for “read-execute-write” component-model semantics, similar to those described in EAST-ADL, making the UPPAAL PORT TA [68] our formalism of choice to specify the intended behavior of the EAST-ADL components. In order to be able to verify the functional and timing requirements at the architectural level, we perform an automatic model-to-model transformation from the EAST-ADL model extended with TA semantics to the input model of UPPAAL PORT. This enables the use of the integrated simulator and of the component-based model checker to formally verify that the architectural model meets its requirements by exhaustively exploring all the possible interleavings of the components in the model [52, 73]. UPPAAL PORT does not perform the usual flattening of the model, and in addition, it is complemented by the Partial Order Reduction Technique (PORT) [10] to improve the efficiency of the analysis.
Even though such techniques (i.e., partial order reduction) attempt to reduce the state space during model checking, the state space explosion is still a real problem when analyzing large industrial-scale systems [66]. To tackle this issue, we incorporate statistical model checking (SMC) techniques in our verification framework, which are supported by the statistical model checking extension of UPPAAL, called UPPAAL SMC [49]. SMC generates stochastic simulations and employs statistical methods to estimate probabilities and probability distributions over time with given confidence levels. For this, we extend the analysis framework with a new transformation that maps the elements of the EAST-ADL functional model into a network of traditional UPPAAL TA [89]. In order to preserve the semantics of the architectural language and for model conciseness, the transformation produces a network of two synchronized TA for each EAST-ADL component, respectively: (i) an Interface TA based the elements provided in the architectural model, and (ii) a Behavior TA that can be further manually edited based on the system requirements or other similar documents. The resulting model enables both exhaustive formal verification for functional and timing properties with the UPPAAL model checker, as well as statistical analysis with UPPAAL SMC. By employing UPPAAL SMC, one can statistically analyze qualitative (e.g., hypothesis testing) and quantitative properties in terms of probabilities and costs. Such a statistical analysis technique can also be applied on high-level design artifacts to provide early information on the resource consumption of the system. Due to the limited resources available to automotive embedded systems (e.g., energy, memory), it is highly desirable to reason about feasibility and worst-case resource consumption of the embedded components before their actual implementation. For this, we take advantage of the EAST-ADL’s resource annotations, and we extend the UPPAAL timed automata model with resource annotations based on the information provided in the architectural model [88]. The end result is a network of priced timed automata that can be analyzed with UPPAAL SMC. The results of this analysis can be seen as valuable feedback on the resource-driven system behavior, prior to the actual implementation.

Testing, the main verification technique used by industry today [13], aims at gaining confidence in the software system through fault detection, that is, observing the differences between the behavior of the implementation and the expected behavior described in the specifications. Testing activities are usually time and resource consuming, and are often conducted by employing ad hoc, error prone, and expensive techniques [96]. This has boosted the development of potentially more efficient testing techniques, like model-based testing [112], 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 [92]. We present and classify some of the most mature tools available at this moment in order 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. By identifying the limitations and existing gaps in this respect, we can also deduce the issues that need to be addressed in order to enhance the applicability of model-based testing techniques. To provide further evidence of the inner workings of different model-based testing tools, we select a set of representative tools that we apply on a simple yet illustrative Coffee/Tea Vending Machine example, to show the differences in modeling notations, test case generation methods, and the produced test cases. These findings have served as basic insight for extending our verification framework with a method for model-based testing against functional and timing requirements, which relies on the same model-checking technique as verification, yet sets the premises for code testing.

Enhancing our verification framework with model-based testing capabilities [91], requires one to define an executable semantics of the EAST-ADL + UPPAAL PORT TA integrated model, as the formal model can be operationally nondeterministic. We propose a way to guide the manual implementation of such formal models, by defining an executable semantics that removes action and timing nondeterminism in a safety-preserving manner. The eventual resulting implementation becomes our system under test (SUT). Next, we show how to automatically generate executable test cases (Python scripts) for the system implementation based on the information provided by their abstract counterparts. The abstract test cases are in turn generated by model checking the EAST-ADL high-level artifacts that are enriched with TA behavior, and previously verified by component-based model-checking techniques. The main goal is to check the feasibility of the EAST-ADL + TA generated abstract test cases by actually running the SUT on the corresponding executable test cases (obtained via our model-based test-case generation), in an attempt to obtain a pass or fail verdict. Therefore, we integrate within a coherent framework EAST-ADL modeling, TA modeling of components behaviors, as well as verification by model checking and abstract test case generation via UPPAAL PORT and UPPAAL, and statistical model checking via UPPAAL SMC.

We validate the proposed framework on an industrial prototype, namely the Brake-by-Wire system provided by AB Volvo. Applying the model-based testing technique on the BBW system yields promising results: all executable
scripts automatically generated from our abstract test cases have end up with a pass or fail verdict, when the code is executed on them. This proves the fact that the abstract test cases contain enough information for the resulting scripts to be suitable for testing code. Exhaustive verification by model checking does not scale when applied on the BBW system, as we are not able to verify the entire system but just a simplified version with only two wheels. To provide a decent assurance of the entire system, we apply statistical model-checking techniques that provide results with given confidence levels. However, in an attempt to ensure exhaustive verification for complex automotive systems, we investigate other analysis techniques that could help reduce the state space in a provably correct way. Dependency analysis has been used previously at implementation level to understand, optimize, or debug software [30]. By applying dependency analysis on architectural models, crucial dependencies can surface earlier in the life cycle. Once computed, these dependencies can be used to prune the architectural models in an attempt to reduce the reachable state spaces during model checking [90]. Assuming a given requirement, our pruning method ensures that only the relevant dependency chains are examined during EAST-ADL model checking against that particular requirement.

A concrete output of this work is our tool called ViTAL [52] that provides automation to our verification and model-based test case generation framework, as it integrates model-checking techniques with EAST-ADL models. In this thesis, the ViTAL tool is applied on a real-world industrial prototype to show the feasibility of our methodology. We deem the BBW use case relevant, due to its complex structure (contains 25 components for sensors, actuators, and computational elements), different triggering patterns (such as periodic and event-based), as well as the diversity of requirements (e.g., functional, timing, resource usage requirements).

1.1 Thesis overview

The thesis is divided in two main parts: (i) a short overview of the research, which includes in Chapter 2 a presentation of the existing methods and tools used throughout the thesis, in Chapter 3 our research goals, in Chapter 4 a description of the research method, in Chapter 5 a brief description of our contributions, in Chapter 6 a discussion of the related work, and in Chapter 7 our final conclusions, and (ii) a collection of six papers that present a detailed description of our research results. The included papers are as follows:

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: I was one of the main contributors of this paper. I worked on developing the formal verification methodology and the ViTAL tool support, together with Eun-Young Kang, Eduard Paul Enoiu, and Cristina Seceleanu. I was also responsible for applying the tool on the industrial use case and presenting the results. Pierre-Yves Schobbens and Paul Pettersson 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 downscaled 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. I developed and presented the transformation from the EAST-ADL model to a network of UPPAAL timed automata. I was also responsible for applying the transformation on the industrial use case and presenting the results. Henrik Kaijser contributed with the transformation of EAST-ADL to Simulink, together with its application on the industrial use case. Marius Mikučionis 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 contributed with useful ideas and comments.


**Abstract:** The growing complexity of modern automotive embedded systems requires new techniques for model-based design that takes under consideration both software and hardware constraints and enables verification at early stages of development. In this context, EAST-ADL was developed as a domain-specific language dedicated to modeling of functional-, software-, and hardware-architecture of automotive embedded systems. This language represents a convenient abstraction when reasoning about the system functionality and supports modeling of relevant extra-functional properties, like timing and resource usage. By providing formal semantics to the EAST-ADL language, as a network of priced timed automata, it becomes possible to reason about feasibility and
worst-case resource consumption of the embedded components. In this paper, we show how to analyze such embedded systems modeled in EAST-ADL by using statistical model checking. We report our experiences from applying this approach to an industrial Brake-by-Wire system.

**Contribution:** I was the main driver of this paper. I extended the UPPAAL timed automata model (introduced in Paper B) with resource annotations based on the information provided in the architectural model, thus creating a network of priced timed automata. I also showed how statistical model checking techniques can be applied to reason about the feasibility and worst-case resource consumption of such automotive embedded systems. The other authors contributed with useful ideas and comments.


**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 the 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. I reviewed the relevant literature and applied the set of selected tools on the illustrative example. Together with Cristina Seceleanu, I wrote the paper and presented the results. Hélène Le Guen provided the description and the application of the MaTeLo tool, and Paul Pettersson contributed with useful ideas and comments on the paper.


**Abstract:** Architectural description languages, e.g. EAST-ADL, provide a comprehensive approach to describing a complex automotive embedded system as a standardized model that encapsulates structural, functional, and extra-functional information. The aim of such model is to enable the system’s documentation, design, early verification, and even code implementation. In this paper, we show how such models can be used one step further, as a basis of a methodology that leads to eventual code verification for embedded systems. Our proposed methodology relies on automated model-based test case generation for both functional and timing requirements, assuming the EAST-ADL architectural model of the embedded system as input. The EAST-ADL model is extended with timed automata semantics, on which model checking is applied to generate abstract test cases for requirements coverage, after which Python test scripts are automatically generated based on the obtained abstract test cases. The Python scripts represent concrete test cases that are then executed on the system implementation to provide test verdicts. The entire methodology is implemented as a toolchain, consisting of the ViTAL and Farkle tools, and is validated on a Brake-by-Wire industrial system prototype.

**Contribution:** I was the main driver of this paper. I developed the abstract test case generation framework and applied it on the industrial use-case. Mehrdad Saadatmand, Andreas Hammar, and Detlef Scholle developed the test conversion and test execution framework and have applied it on the industrial use case. The other authors contributed with useful ideas and comments.

Abstract: Dependency analysis techniques are widely used to understand software implementations, and reduce their verification efforts. Recently, architectural languages have started to be integrated in the development of complex embedded systems. Such languages provide early development artifacts, which can be used to specify the structure and functionality of a system, and can be also analyzed in order to provide early information regarding the system’s correctness. By performing dependency analysis on architectural languages, crucial dependencies can surface earlier in the life cycle. Once computed, these dependencies can be used to prune the architectural models in an attempt to reduce the early design-stage verification efforts. In this paper, we propose a dependency analysis-based technique that can be applied to prune models in EAST-ADL, an architectural description language tailored to automotive systems development. To achieve correct pruning, we investigate the types of dependencies that can appear in an architectural model, and how these dependencies create dependency chains within the model. Next, we investigate how such dependency chains can be exploited in formal verification, to reduce verified state-spaces during model checking. Assuming a given requirement, our pruning method entails that only the relevant dependency chains are examined during EAST-ADL model checking against that particular requirement. We validate our analysis results by comparing them to those obtained by applying an analytical approach for end-to-end timing analysis in EAST-ADL models. The methodology is illustrated on a Brake-by-Wire industrial system.

Contribution: I was the main driver of this paper. I developed the dependency analysis framework and the pruning method of the architectural models. I was also responsible for applying the tool on the industrial use case and presenting the results. Saad Mubeen provided the analytical approach for end-to-end timing analysis and Cristina Seceleanu contributed with useful ideas and comments.
Chapter 2

Preliminaries

In this chapter, we introduce important technical concepts used throughout the thesis. We start by presenting a short overview of model-driven development in Section 2.1, and the E AST-ADL architectural language in Section 2.1. In Section 2.3, we provide a short description of the analysis and verification methods used in the thesis, which include symbolic simulation, model checking, statistical model checking, and model-based testing, together with the U PPAAL toolchain. We end this chapter with an overview of the dependency analysis method targeting architectural models, in Section 2.3.5.

2.1 Model-driven Development

Models, that is, abstractions of systems, are early design artifacts closely coupled to the problem domain and loosely coupled with the underlying implementation technology, making them easier to specify, understand, and maintain than traditional code. Model-driven development [107] takes advantage of this and proposes a development paradigm that focuses on models rather than the code. To be useful and effective, a model must possess five key characteristics, as follows:

• Abstraction: a model is a reduced rendering of the system, by removing or hiding details that are irrelevant for a given viewpoint;

• Understandability: a model must present the information in an expressive manner that appeals to the intuition;
Chapter 2

Preliminaries

In this chapter, we introduce important technical concepts used throughout the thesis. We start by presenting a short overview of model-driven development in Section 2.1, and the EAST-ADL architectural language in Section 2.1. In Section 2.3, we provide a short description of the analysis and verification methods used in the thesis, which include symbolic simulation, model checking, statistical model checking, and model-based testing, together with the UPPAAL toolchain. We end this chapter with an overview of the dependency analysis method targeting architectural models, in Section 2.3.5.

2.1 Model-driven Development

Models, that is, abstractions of systems, are early design artifacts closely coupled to the problem domain and loosely coupled with the underlying implementation technology, making them easier to specify, understand, and maintain than traditional code. *Model-driven development* [107] takes advantage of this and proposes a development paradigm that focuses on *models* rather than the code. To be useful and effective, a model must possess five key characteristics, as follows:

- **Abstraction**: a model is a reduced rendering of the system, by removing or hiding details that are irrelevant for a given viewpoint;

- **Understandability**: a model must present the information in an expressive manner that appeals to the intuition;
• **Accuracy**: the model must provide a true representation of the features of interest;

• **Predictiveness**: the model should enable the prediction of relevant and non-obvious properties through verification;

• **Inexpensive**: The model must be significantly cheaper to construct and verify than the actual system.

In model-driven development, models are viewed as more than merely documentation. Assuming that the modeling language takes on the role of the implementation language, then models can be used for automatic code generation [37]. Such practices, which could boost productivity and reliability, are enabled by the advances in automation technologies and by the newly emerged industry-wide standards like the Unified Modeling Language (UML) [105]. Automatic code generation is often coupled with automatic model verification [43], in an attempt to ensure the presence of desirable properties and the absence of undesirable ones in the model. This verification can take many different forms, such as formal analysis (e.g., performance analysis based on queuing theory or checking safety and liveness properties), or simulating the models as an empirical approach to verification. In both cases, it is highly desirable to achieve this on highly abstract and incomplete models that arise early in the development cycle, because this is when software designers take some of the fundamental design decisions.

### 2.2 Architectural Modeling: EAST-ADL

The architecture of a system is defined as “the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution” [87]. *Architectural description languages* (ADL) provide a framework for designing and specifying the structure of the overall system, which includes gross organization and global control structure, protocols for communication, synchronization and data access, assignment of functionality to design elements, physical distribution, scaling and performance, and even selection among design alternatives [63]. In their book, “Software Architecture: Perspectives on an Emerging Discipline” [108], Garlan and Shaw promote software architecture concepts such as components, connectors, and styles. Therefore, the main
elements of an ADL are: (i) components, primitive building blocks, (ii) connectors, mechanisms of combining components, and (iii) architectural configurations, rules for referring to the combination of components and connectors.

Figure 2.1: The EAST-ADL’s levels of abstraction.

Several architecture description languages have been proposed over the years for modeling software/systems [94], such as AADL [54], ACME [61], Rapide [84], etc. We focus on EAST-ADL [33], an emerging architecture description language dedicated to the automotive domain and aligned with AUTOSAR [56] that is an open automotive industry standard between suppliers and manufacturers. EAST-ADL provides a standardized format that captures different aspects of the automotive electronic system, like vehicle features, functions, requirements, software and hardware components, and communication. As shown in Figure 2.1, the system is defined at four levels of abstraction, as follows:

- **Vehicle Level** describes the electronic system features as they are perceived externally, which also might include the system requirements and their allocation on a particular feature;

- **Analysis Level** provides an abstract functional architecture defining the embedded system from a functional point of view, without prescribing a specific hardware topology;
• **Design Level** provides a detailed functional architecture that can be extended with timing properties (e.g., triggering periods, the execution times, end-to-end deadlines), introduces the hardware architecture, and describes the allocation of the software components onto the hardware architecture;

• **Implementation Level** provides the implementation of the embedded system represented using AUTOSAR elements.

At each abstraction level, the architectural model relies on the definition of a set of *FunctionTypes*. Each *FunctionType* has a set of associated modeling elements that includes: (i) a set of *FlowPorts* to receive and provide data, (ii) a *FunctionTrigger* that can be either event driven or time driven, and (iii) a *FunctionBehavior* that is a transfer function performing the data computations and defined using external notations and tools (e.g. Simulink). The execution of each *FunctionType* is done based on synchronous execution semantics (also known as “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.

At each level of abstraction, the system is modeled as a set of interconnected *FunctionPrototypes*, where each *FunctionPrototype* is an instantiation of the corresponding *FunctionType*. The system model can be extended with annotations for orthogonal aspects like requirements, timing properties, generic constraints, etc. *EAST-ADL* also provides means to plan different validation and verification activities for different levels of abstraction.

To exemplify these concepts, we show a small, simplified automotive system in *EAST-ADL*. As depicted in Figure 2.2 (a), the system has only two features that are presented at Vehicle Level: (i) the *Cruise Control* and (ii) the *Braking Control*. At this abstraction level, system requirements connected to a particular system feature can be defined. In case of *Braking Control*, two such requirements have been defined, *Req#1* connected to the time to standstill of the vehicle, and *Req#2* connected to the anti-lock braking function. At the Analysis Level, the components of the system are introduced. As depicted in Figure 2.2 (b), the architecture consists of three *FunctionPrototypes*: (i) *Sensor_In*, of type *Sensor*, (ii) *Comp_Block* of type *Computation*, and (iii) *Actuator_Out* of type *Actuator*. At Design Level, the model is further detailed. As shown in Figure 2.2 (c), the *Comp_Block* is decomposed in two *FunctionPrototypes*, namely the *BrakeCalculator* and the *ABSFunction*. At this abstraction level, the architectural model contains information regarding the triggering patterns (i.e., time-triggered components with the period defined by the *Periodic_Constraint*)
and the timing behavior (modeled as \textit{Exec\_Time\_Constraint} describing the time needed for the component to compute the output, and the \textit{Reactive\_Constraints} describing the maximum time needed for the data to propagate throughout the system). We do not provide an overview of the system at Implementation Level since at this level of abstraction \textit{EAST-ADL} uses AUTOSAR elements, for which we refer the reader to the relevant literature [56].

### 2.2.1 Industrial Use Case: The Brake-by-Wire System

In this section we introduce the Brake-by-Wire (BBW) system, an industrial prototype developed by AB Volvo, which is used throughout this thesis as a running example to illustrate our techniques and results. The BBW system 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 brake torque of each wheel, as follows:

\[
wheel\_torque = (pos/100) \times max\_Brake\_Torque \times distribution \quad (2.1)
\]

In equation 2.1, \(pos \in [0,100]\) is the position of the brake pedal, \(max\_Brake\_Torque\) is the maximum global brake torque, and \(distribution\) is the static distribution factor. At each wheel, a sensor measures its speed, which is used by the ABS algorithm together with the wheel torque to compute the actual brake torque that is sent to the actuator. The ABS algorithm computes the slip rate \(s\), as follows:

\[
slip\_rate = (vehicle\_speed - wheel\_speed \times R) / vehicle\_speed \quad (2.2)
\]

In equation 2.2, \(vehicle\_speed\) is the speed of the vehicle, \(wheel\_speed\) is the speed of the wheel, and \(R\) is the radius of the wheel. The friction coefficient has a nonlinear relationship with the slip rate: when \(slip\_rate\) starts increasing, the friction coefficient also increases, and its value reaches the peak when \(slip\_rate\) is around 0.2. After that, any further increase in \(slip\_rate\) reduces the friction coefficient of the wheel. For this reason, if the \(slip\_rate\) is greater than 0.2 the brake actuator is released and no brake is applied, otherwise the requested brake torque is used.

Figure 2.3 depicts the functional architecture of the BBW system at Design Level, extended 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 PeriodConstraint. For instance, each of the four ABS FunctionPrototypes are triggered every 10 ms, and their execution takes at most 0.64 ms according to the associated ExecTimeConstraint. All the sensors and the actuators are event triggered, meaning that their 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 functionality of the system is realized by: (i) sensors (e.g., pLDM Sensors), extended with annotations for timing properties like triggering period and execution time. The brake actuator is released and no brake is applied, otherwise the requested torque is sent to the actuator. The ABS algorithm computes the slip rate, and its value reaches the peak when the friction coefficient also increases, and its value reaches the peak when slip rate starts increasing, the distribution factor. At each wheel, a sensor measures its speed, which is used by the ABS algorithm together with the wheel torque to compute the actual brake distribution factor. The ABS algorithm together with the wheel torque to compute the actual brake distribution factor. The brake actuator is released and no brake is applied, otherwise the requested brake torque is sent to the actuator. The ABS algorithm computes the slip rate, and its value reaches the peak when the friction coefficient also increases, and its value reaches the peak when slip rate starts increasing, the distribution factor. At each wheel, a sensor measures its speed, which is used by the ABS algorithm together with the wheel torque to compute the actual brake distribution factor.

In equation 2.2, \( R \) speed of the wheel, and \( \omega \) is the position of the brake pedal, which is used to compute the brake torque to the brake pedal reads its position, which is used to compute the brake torque as follows:

\[
\text{Torque} = \text{maxBrakeTorque} \times \frac{\text{pos} - \text{pos}/[0,100]}{\text{maxBrakeTorque}}
\]

In equation 2.1, \( s \) = \( \frac{\text{pos} - \text{pos}/[0,100]}{\text{maxBrakeTorque}} \) is the position of the brake pedal, \( \omega \) is the speed of the vehicle, \( \text{wheel torque} \) is the maximum global brake torque, and \( \text{slip rate} \) is the slip rate of each wheel, as follows:

\[
\text{slip rate} = \frac{\text{wheel speed} \times \text{wheel torque}}{\text{vehiclespeed} \times \text{maxBrakeTorque}}
\]

Figure 2.3 depicts the functional architecture of the BBW system at Design Level.
The BBW system has a set of requirements that have to be verified in order to ensure that they are met already at the architectural level. In Table 2.1, we present some of the requirements of the BBW system, expressed in natural language:

| Functional | R₁: If the slip rate is greater than 0.2, then ABSBrakeTorqueOut shall be set to 0 Nm. |
|           | R₂: If slip rate is smaller or equal to 0.2, then ABSBrakeTorqueOut shall be set to RequestedTorqueIn. |
|           | R₃: If RequestedTorqueIn is 0, then ABSBrakeTorqueOut shall be set to 0 Nm. |
|           | R₄: DriverReqTorqueOut shall be set to a value proportional to BrakePedalPositionIn with respect to its max value. |
|           | R₅: If BrakePedalPositionIn has a value higher than its max value, then DriverReqTorqueOut shall be set to its max value. |
|           | R₆: VehicleSpeedEstOut shall be set to the average value of WheelSpeed_FLIN, WheelSpeed_FRIn, WheelSpeed_RLIN and WheelSpeed_RRIn |
| Timing    | R₇: The time needed for a brake request to propagate through the system, from the brake pedal sensor to the wheel actuator, should not exceed 200 ms. |
|           | R₈: The time needed for a brake request to be computed, (i.e., from the brake pedal sensor to the Global Brake Controller), should not exceed 130 ms. |
|           | R₉: The time needed for a wheel to process the brake request, from the ABS to the wheel actuator, should not exceed 70 ms. |
|           | R₁₀: The synchronization delay (i.e., the time needed for a brake request to propagate from the brake pedal sensor to two distinct wheel actuators), should not exceed 20 ms. |

Table 2.1: The BBW system requirements.

2.3 Methods for System Analysis and Verification

Verification [57, 117] aims at checking and providing objective evidence that a product, service, or system meets its set of requirements. In the development phase, analysis and verification procedures can involve the simulation or the
application of exhaustive or statistical formal methods to analyze the system, as a whole or in terms of subsystems. In the post-development phase, verification involves the use of testing methods to ensure that the system continues to meet the initial design requirements. In the following, we overview the analysis and verification methods that are employed in this thesis.

2.3 Methods for System Analysis and Verification

2.3.1 Symbolic Simulation

Simulation [81], in the broadest sense, means mimicking the operation of a process or a system over time. In computer science, a simulation is the computation of a given execution of some model of the system. Simulation is often used in industry to verify software or hardware systems. This is enabled by the fact that such models (i.e., for computer programs or logic hardware designs) often have an operational semantics that can be used directly for simulation.

In symbolic simulation [36], a technique introduced by King [76], multiple possible executions of a system are considered simultaneously. This is achieved through symbolic variables that provide such multiple executions in a simulation state representation. For instance, in a symbolic simulation of a gate design, inputs may be parametrized by (Boolean) variables, and outputs are functions of these variables. The result of a simulation is a sequence of concrete system states, where the variables values are represented by all possible valuations within a state. Because symbolic simulation can cover many system executions in a single simulation, it can greatly reduce the size of the verification problem. In this thesis, we use symbolic simulation to verify automotive systems that have been originally described in EAST-ADL [73].

2.3.2 Model Checking

Formal verification of system models [118] is the process of rigorously exploring the behavior of the model expressed in an abstract mathematical notation, in order to decide whether the system’s design meets specified requirements. A popular formal verification technique is model checking [44], which relies on efficient algorithms for exhaustively exploring finite-state system models, in order to check whether the model satisfies given requirements specified most often in temporal logics [51, 43, 101]. Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula \( p \), and a model \( M \) with initial state \( s_0 \), decide if \( M \) satisfies \( p \), starting from \( s_0 \): \( M, s_0 \models p \). As depicted in Figure 2.4, the model checker reduces the verification to a reachability problem or to a temporal logic verification by algorith-
mically 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 a formalized safety requirement is not satisfied, the model checker can provide a counter-example). Model checking has been widely used to verify industrial systems [3, 15, 18, 19, 24, 42].

![Figure 2.4: Verification based on model-checking techniques](image)

The system models are often expressed as automata, out of which timed automata is a popular formalism for modeling real-time systems [12], supported by the UPPAAL tool suite [22]. In timed automata, the system’s behavior is represented as a timed graph consisting of a finite set of locations and a finite set of labeled edges that connect the locations. The graph is extended with real-valued timing constraints that use real-valued variables called clocks, which measure the elapse of time. The clock variables are initialized with zero when the system is started, and then increase synchronously at the same rate [23]. Constraints on the clock variables represent guards that are assigned on the edges, which are used to restrict the behavior of the automaton. Actions and invariant conditions, which are boolean constraints on clocks that set upper bounds on the time that the automaton is allowed to delay in a location, are used to enforce progress properties. To model concurrent systems, timed automata uses parallel composition that allows interleaving of actions as well as handshake synchronizations.

Timed automata composed in parallel form a network of timed automata, which 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 the clock variables. The transitions from one state to another can be: (i) discrete transitions, corresponding to traversing an edge whose guard is satisfied, or (ii) delay transi-
tions, with all clocks in the network being incremented with the same value.

We use a simple coffee vending machine to present different timed automata variants used by the UPPAAL model checker, and the UPPAAL PORT model checker - an extension of UPPAAL for component-based systems. 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.** UPPAAL is an integrated tool environment for modeling, simulation, and model checking of real-time systems described as networks of TA. The UPPAAL TA [17] is defined as a tuple:

\[
TA \triangleq (L, l_0, C, A, E, I)
\]

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 \( r \)-actions, \( E \subseteq L \times A \times B(C) \times 2^C \times L \) is a set of edges between locations with an action, a guard, and a set of clocks to be reset, where \( B(C) \) defines a set of conjunctions over simple conditions of the form \( x \triangleright c \) or \( x - y \triangleright c \) where \( x, y \in C, c \in \mathbb{N} \) and \( \triangleright \in \{<, \leq, =, \geq, \geq \} \), and \( I : L \rightarrow B_{\text{upper}}(C) \) is a function that assigns invariants to locations, where \( B_{\text{upper}}(C) \subseteq B(C) \) denotes a subset of clock constraints resulted from the restriction to upper bounds \( \ll \in \{<, \leq \} \), where a clock constraint \( b \) is defined according to the grammar \( b := \text{true} \mid x \triangleright c \mid b_1 \land b_2 \). To denote an edge from location \( l \) to \( l' \), with guard \( g \), update action \( a \), and clock resets \( r \), we write \( l \xrightarrow{g,a,r} l' \), for \((l, g, a, r, l') \in E\). In UPPAAL a location can be marked as urgent (marked with an \( u \)) or committed (marked with a \( c \)) indicating that the time cannot progress in such locations. The latter is a more restrictive, indicating that the next edge to be transversed needs to start from a committed location. The communication is modeled via synchronization channels (e.g., channel! and channel?) with rendezvous or broadcast semantics. In the rendezvous communication, a sender (i.e., channel!) synchronizes with a receiver (i.e., channel?), provided that the sending and receiving edges are enabled, that is, their guards are satisfied. In the broadcast semantics, a sender can synchronize with an arbitrary number of receivers, and if cannot synchronize with any receiver, then the sender can still execute the channel! action, since the broadcast sending is never blocking, as compared to the rendezvous one.

Figure 2.5 depicts the Coffee Vending Machine modeled as a network of two UPPAAL TA: the User TA depicted in Figure 2.5 (a), and the Coffee Maker
26 Chapter 2. Preliminaries

Figure 2.5: The Coffee Vending Machine example in UPPAAL.

(a) User TA.

(b) Coffee Maker TA.

TA depicted in Figure 2.5 (b). The user can insert a number of coins, which is modeled as an edge on which the \( \text{coin} \) variable is increased. Next, the user presses the button modeled as a synchronization channel named \( \text{button} \) that triggers the location change from \( \text{Init} \) to \( \text{Prepare} \) in the Coffee Maker TA, provided that the user has inserted three coins such that the guard \( \text{coin} == 3 \) evaluates to true. The Coffee Maker TA can be in the \( \text{Prepare} \) location between 5 and 15 time units, which is modeled as an invariant \( x < 15 \) and a guard \( x > 5 \) on clock \( x \). The location change to \( \text{Init} \) in the Coffee Maker TA, also triggers the location change to \( \text{Init} \) in the User TA based on the \( \text{reset} \) synchronization channel.

UPPAAL [17] provides formal verification for timed systems based on the above TA formalism. The tool uses symbolic semantics and symbolic reachability techniques to analyze dense-time state spaces against properties formalized as a subset of (timed) computation tree logic ((T)CTL) [11].

UPPAAL PORT. UPPAAL PORT [69] is a verification tool for component-based systems implemented on top of the traditional UPPAAL tool. 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:

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

(2.4)

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 reset, and $I$ assigns clock invariants to locations.

An 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 2.6 depicts the Coffee Vending Machine modeled in UPPAAL. The system model is presented in Figure 2.6 (a), where data ports, both input and output, are marked with a square, while the trigger ports are marked with a triangle. The model 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 $\$ coin$, and the component executes the behavior according to the TA model. As de-
picted in Figure 2.6 (b), if the user has introduced three coins, then the guard \$\text{coin} == 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. As depicted in Figure 2.6 (c), 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 variable is set to 1, and this data is provided to the data output port.

UPPAAL PORT takes as input the model described above, to provide formal verification of the system without the usual flattening of the TA network [68]. 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 [20] to increase independence, being suited for the analysis of “read-execute-write” component models.

In this thesis, we use model checking to exhaustively verify automotive systems that have been originally described in EAST-ADL with both the UPPAAL [89] and UPPAAL PORT [73] model checkers.

2.3.3 Statistical Model Checking

Traditional model checking can be subject to combinatorial explosion (that is, the state space of the model grows exponentially with the number of reachable states, and the transitions that the model can make between these states) [114]. Statistical model checking (SMC) [82] has been proposed as an alternative that avoids exhaustive exploration of the state space of the model during verification. SMC is especially useful in analyzing qualitative and quantitative properties of stochastic systems. The core idea of SMC is to monitor different simulations of the system, and then use results from statistics (including sequential hypothesis testing or Monte Carlo simulation) in order to decide whether the system satisfies the property with some degree of confidence. As compared to exhaustive symbolic model checking, which is an exact verification method, statistical model checking based on simulation cannot guarantee the correctness of the result. However, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory and time intensive than numerical ones (or than traditional model checking), and are sometimes the only option [120].

We use the same coffee vending machine description introduced in Section
2.3.2 to present UPPAAL SMC - an extension of the UPPAAL tool for statistical model checking (SMC).

**UPPAAL SMC.** To enable statistical model checking with UPPAAL SMC [39], the model is represented as a network of priced timed automata (PTA). A PTA is defined as a tuple:

\[
PTA = \langle L, l_0, X, E, F, I \rangle
\]

where \(L\) is a finite set of locations, \(l_0 \in L\) is the initial location, \(X\) is a finite set of continuous variables, \(\Sigma = \Sigma_i \cup \Sigma_o\) is a finite set of actions partitioned into inputs (\(\Sigma_i\)) and outputs (\(\Sigma_o\)), \(E\) is a finite set of edges of the form \((l, g, a, \varphi, l')\), where \(l\) and \(l'\) are locations, \(g\) is a predicate on \(\mathbb{R}^X\), action label \(a \in \Sigma\), and \(\varphi\) is a binary relation on \(\mathbb{R}^X\), \(F(l)\) a delay function for a location \(l \in L\), and \(I\) assigns an invariant predicate \(I(l)\) to any location \(l\). With this definition, UPPAAL SMC extends the TA tuple used by UPPAAL with the delay function \(F\) that allows the continuous variables to evolve according to linear differential equations\(^1\). This enables the modeling of cost variables that can evolve at (positive) integer rates and that are used in this thesis to capture the system’s resource usage. In UPPAAL SMC, the automata have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or user-defined exponential distributions for unbounded delays.

UPPAAL SMC allows one to specify an arbitrary rate for the continuous variables on any location, 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 to allow the tool to compute nontrivial functions using small step integration. For more details and examples on the modeling formalism used by UPPAAL SMC, we refer to the work of Bulychev et al. [39].

Figure 2.7 depicts the Coffee Vending Machine modeled in UPPAAL SMC. It extends the model shown in Figure 2.5 with the following elements: (i) a continuous variable `energy` that increases with the rate of consumption `energy' == 3 for location `Init`, and `energy' == 15` for location `Prepare` (i.e., the energy consumed by the machine to prepare the coffee increases linearly, at rate 15),

\(^1\)Actually, the language is more expressive than this, allowing the modeling of nonlinear differential equations, but in this thesis we restrict our models to linear differential equations.
and (ii) a rate of exponential of 1 for location Init, which means that the automaton could potentially stay in the location forever, but it will stay there on average for 1 time unit. As shown in Figure 2.7 (b), the set of conjunctions is no longer bounded to \( c \in \mathbb{N} \), but to \( c \in \mathbb{R} \).

UPPAAL SMC extends the UPPAAL model checker with statistical model checking (SMC) algorithms [50] to decide qualitative properties and compute quantitative 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 [48].

UPPAAL SMC uses an extension of the Weighted Metric Temporal Logic (WMTL) [38] to verify properties like:

- **Hypothesis testing**, e.g., check if the probability to reach state \( \phi \) within cost \( x \leq C \) is greater or equal to a certain threshold \( p \), \( Pr(\Diamond x \leq C \phi) \geq p \),

- **Probability evaluation**, e.g., calculate the probability \( Pr(\Diamond x \leq C \phi) \) for a
given network of PTA,

- **Probability comparison**, e.g., is $P(\diamond x \leq C \phi_1) > P(\diamond y \leq D \phi_2)$?

The first and the third of the properties shown above are qualitative, whereas probability evaluation is quantitative.

In this thesis, we use UPPAAL SMC to verify different functional and timing properties that cannot be verified through exhaustive verification due to the state-space explosion problem. We also employ UPPAAL SMC to reason about feasibility and worst-case resource consumption of automotive embedded systems.

### 2.3.4 Model-based Testing

Testing is the primary verification technique used by industry to evaluate the quality of a software or hardware product and to improve it by identifying possible defects and problems [112, 113]. It consists of the verification of a finite set of executions against the expected outcome. A **test case** is defined as the combination of a test input values and the expected outcomes of the system, and a collection of such test cases is called a **test suite**. During the execution of the system under test (SUT) on the test suite, failures (i.e., undesired behaviors) can be observed. A **fault** (i.e., an error in the system) is the root of the failure and it can stem from the specification, design, or the actual implementation.

![Figure 2.8: A general model-based testing process.](image)

Model-based testing [113] is a testing technique that attempts to automate the generation and execution of test cases, based on an abstract **model** of the system. Since testing is a technique that aims to identify errors, we consider it a verification technique, even though it is not an exhaustive verification technique, like model checking. In model-based testing, the system implementation is abstracted into a model that includes only the aspects of the system that are intended to be tested (Step 1 in Figure 2.8). The testing process is automated...
based on the definition of a test selection criterion (Step 2 in Figure 2.8), a specification of the features of the system to be tested that selects the desired test cases from an infinite number of possible tests (Step 3 in Figure 2.8). Such a test selection criterion most often refers to a certain functional requirement of the system, to the structure of the model, or to different stochastic characteristics. These test cases are often called abstract test cases because they reflect only the information encoded in the model, therefore they are independent of the actual implementation (Step 4 in Figure 2.8), so they can not be directly executed on the SUT.

To enable test execution, one needs to add to the abstract test cases all the details of the system implementation that are not mentioned in the abstract model (Step 5 in Figure 2.8). This can be done with the help of a transformation tool, or more often by writing an adaptor that wraps around the SUT and implements each abstract operation in terms of the low-level system facilities.

The SUT is run on the concrete test cases (Step 6 in Figure 2.8). At the end of each test execution, a verdict is returned, which represents the result of comparing the output produced by the SUT with the expected output provided by the concrete test case generated from the abstract test case. 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. Besides the testing verdict, the testing tool can return additional, relevant information, which includes code coverage, defect localization, etc.

2.3.5 Dependency Analysis of Architectural Models

Dependency chains represent underlying dependency relationships between components in the architectural model, based on the connections between components and the associated constraints of their interactions. Since architectural models are available early in the development cycle, the identified dependency chains provide a basis for reasoning about the existing system dependency before its implementation. Assuming that a precise mapping is maintained between the architectural description and the actual implementation, the dependency analysis can identify crucial dependencies earlier in the life cycle, which would create cost overruns and degrade the quality of the systems, if left unmanaged.

Several research endeavors have focused on the dependency analysis of architectural models [83, 111, 115, 122]. The core idea of these methods lies in identifying “links”, dependency relationships within the architecture, which
directly connect elements of the model and produce a chain of dependencies that can be followed during analysis. Stafford et al. [111] have identified three types of dependency chains that can be found in an architectural model:

- **Affected-by** chains consisting of a set of architectural elements that could affect the behavior of another element;
- **Affects** chains consisting of a set of architectural elements that could be impacted by a change in another component;
- **Related** chains consisting of a set of architectural elements that may affect or may be affected-by another element.

All these relationships can involve the structure or the behavior of the model. Structural relationships can locate source specifications that contribute to the description of a state or interaction, while behavioral dependencies relate different states or interactions to each other. However, the particular kinds of chains that can be found in an architectural model are highly dependent on the architectural language used.

To uncover the existing dependency chains, a matrix representation of the model is required. The matrix is \( m \times n \), where:

- \( m \) represents the number of ports in the model together with any events generated in the environment, and
- \( n \) represents the number of ports in the architecture.

In the matrix, the rows represent the source of the dependency, while the columns represent the destination of the dependency. The relationships associated with the inter- and intra component connection produce dependence links. The existing dependency links are annotated in the matrix and an algorithm can be used to compute the dependency chains in the model, chains that can be used during analysis to answer important questions about the model.

In this thesis, we compute the dependencies present in an EAST-ADL architectural model and we use these dependencies to prune the architectural models in an attempt to reduce the early design-stage verification efforts.
Chapter 3

Research Focus

In this chapter we present the main problems addressed in this thesis (see Section 3.1) and introduce our main research goals (see Section 3.2).

3.1 Problem Description

Architectural description languages, such as EAST-ADL, can be introduced early in the development process to describe complex automotive systems as component-based systems. At system level, EAST-ADL relies on a well-defined and standardized structure that integrates the functional and extra-functional information (e.g., timing properties, resource consumption) of complex, safety-critical automotive systems, up to the system implementation that is compatible to the AUTOSAR standard [1]. The choice of a system's architecture has impact on the eventual implementation, with respect to performance, manageability etc. Such models should then be reasoned about, in order to establish system properties early in its development cycle.

To enjoy the fully-fledged advantages of reasoning, such architectural models could benefit from a verification framework that includes, ideally, both formal analysis and model-based testing. Rigorous verification techniques applied on system architectural models lead to gaining indications of the correctness of the design with respect to both functional and extra-functional requirements, whereas model-based testing techniques ensure that the final implementation conforms with the architectural model. Such a verification framework is endorsed by emerging safety standards, like ISO 26262 [99], which place strict...
Chapter 3

Research Focus

In this chapter we present the main problems addressed in this thesis (see Section 3.1) and introduce our main research goals (see Section 3.2).

3.1 Problem Description

Architectural description languages, such as EAST-ADL, can be introduced early in the development process to describe complex automotive systems as component-based systems. At system level, EAST-ADL relies on a well-defined and standardized structure that integrates the functional and extra-functional information (e.g., timing properties, resource consumption) of complex, safety-critical automotive systems, up to the system implementation that is compatible to the AUTOSAR standard [1]. The choice of a system’s architecture has impact on the eventual implementation, with respect to performance, manageability etc. Such models should then be reasoned about, in order to establish system properties early in its development cycle.

To enjoy the fully-fledged advantages of reasoning, such architectural models could benefit from a verification framework that includes, ideally, both formal analysis and model-based testing. Rigorous verification techniques applied on system architectural models lead to gaining indications of the correctness of the design with respect to both functional and extra-functional requirements, whereas model-based testing techniques ensure that the final implementation conforms with the architectural model. Such a verification framework is endorsed by emerging safety standards, like ISO 26262 [99], which place strict
requirements on the functional safety of the system in an attempt to guarantee the quality of each level of design abstraction. A popular and effective formal verification technique is model checking, which can be used both for ensuring that all possible system behaviors meet the requirements (functional, timing etc.), and also for generating test cases automatically. However, the complexity of the automotive systems often leads to state-space explosion during model checking, thus limiting the adoption of such methods in industrial practice. Moreover, such methods are not often integrated with the architectural modeling, and this is in fact the case for EAST-ADL.

In this thesis, we investigate the use of model checking techniques to supply a verification framework tailored for EAST-ADL models. This research problem raises various challenges. First, we need to consider the particularities of the architectural language, such as the component-related functional annotations, the extra-functional annotations, the use of external tools to describe the behavior of the components, but also the lack of formal semantics of the language. The latter implies that we need to define a formal semantics of EAST-ADL, as a prerequisite to creating formal models that can be model checked both for verification and test case generation. Second, in case of exhaustive verification of EAST-ADL models, we also need to investigate possible means to achieve scalable verification (e.g., component-aware model checking, model pruning based on dependency analysis, etc.), in an attempt to overcome the potential state-space explosion problem. Third, in order to obtain a complete model-based testing framework, we also need to convert the generated test cases into executable scripts which the SUT can be run on, in order to produce valid test verdicts with respect to functional or timing requirements coverage criteria. In Section 3.2 we present our research goals that directly address the challenges described in this section.

3.2 Research Goals

Providing a formal analysis and verification framework for architectural models like EAST-ADL, for which only informal semantics is available, while attempting to tackle the scalability of the verification, is a challenging task. The issues described in Section 3.1 lead to the formulation of the main goal of the thesis, as follows:

**Overall Goal.** Investigate how model-checking techniques can be used for the architecture-centric analysis, verification, and model-based testing of automo-
This overall goal is too abstract and too wide to be directly addressed, so we have further divided it into six more concrete and narrower subgoals. Since our overall goal refers to employing model checking to analyze architectural models of automotive systems, and we chose to focus on the EAST-ADL language, we need to lay the foundation that enables the application of formal techniques to EAST-ADL. This is achieved by defining a formal semantics of the architectural language, which supports all the relevant information, such as dependencies between different system components, triggering patterns, timing annotations, resource consumption annotations, etc. Hence, we formulate the first subgoal as follows:

**RG 1. Define a formal semantics of the EAST-ADL architectural models.**

Tackling the above subgoal enables the formal analysis of EAST-ADL architectural models extended with formal semantics, based on model-checking techniques, hence giving rise to our second subgoal as follows:

**RG 2. Provide analysis and verification support based on model-checking techniques, to EAST-ADL models extended with formal semantics.**

At this point, we are able to partly address the overall goal. Facilitating early verification of EAST-ADL models is beneficial, but not sufficient, since we also need to ensure that requirements are met by the implementation of the respective structural description. A straightforward way to achieve this is to employ architecture-based testing techniques that hopefully help us reach our target goal. A first step towards meeting such a desideratum is to investigate the state of the art in model-based testing of requirements-based system specifications, to get insight into commonalities, variabilities, strengths and weaknesses of the existing solutions. This justifies our third subgoal stated below:

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

We address the above subgoal by performing a literature review of the state of the art and state of practice in model-based testing of requirements-based system specifications. The identified gaps in the current tools, as well as the lack of model-based testing methods that use an architectural model as the input artifact have motivated us to extend our formal verification framework with model-based testing capabilities. Therefore, our fourth subgoal is defined as follows:
RG 4. Develop a framework that integrates model checking and model-based testing, starting from EAST-ADL system descriptions.

Such a framework enables both the formal verification and model-based testing of automotive embedded systems that are described in the EAST-ADL architectural language. However, the complexity of automotive systems might hinder the formal verification of the system due to the state-space explosion problem. Dependency analysis can be applied at the architectural level to identify crucial dependencies early in the life cycle. These dependencies can be exploited during verification, in an attempt to reduce the size of the model and verify only the relevant subsystems with respect to a certain system requirement. This gives rise to the fifth subgoal, as follows:

RG 5. Develop a pruning method for EAST-ADL architectural models based on intra- and inter-component dependency analysis.

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

RG 6. Validate the methodology on an industrial system.
Chapter 4

Research Methodology

Research, which can be equated with the search for new knowledge, is a scientific and systematic inquiry for relevant information, and it can be classified into several classes [77]:

- **Applied vs. Fundamental**: fundamental research is concerned with the formulation of theories, and their generalizations, whereas applied research attempts to find solutions to immediate problems encountered by an industrial or business organization;

- **Descriptive vs. Analytical**: descriptive research provides a description of the state of affairs at the current moment through surveys and other fact-finding enquirers, while analytical research provides a critical evaluation by analyzing the facts and information already available;

- **Quantitative vs. Qualitative**: quantitative research is applied to a phenomenon that can be expressed in terms of quantity (i.e., the measurement of quantity or amount), while qualitative research is applied to a qualitative phenomenon (i.e., phenomena relating to or involving quality or kind);

- **Conceptual vs. Empirical**: conceptual research is related to some abstract idea(s) or theory, and used to develop new concepts or to reinterpret existing ones, while empirical research is a data-based research, coming up with conclusions that are capable of being verified by observation or experiment.
The research process is achieved through performing a series of activities, such as: (i) defining and redefining problems, (ii) review the relevant literature, (iii) formulating hypotheses and suggesting solutions, (iv) collecting, organizing and evaluating data, (v) making deductions and reaching conclusions, and (vi) testing the conclusions to determine whether they support the hypothesis. In this context, the research methods are defined as concrete techniques that are used to solve a given research problem. They include methods for the collection of data (e.g., data is already available but not sufficient to arrive at the required solution), statistical techniques (e.g., establishing relationships between the data and the unknown), and the evaluation of the accuracy of the results obtained. 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. In computer science, the research process can be described by fours major steps [71], as depicted in Figure 4.1.

Our research goal is to investigate how model-checking techniques can be used for the verification of automotive embedded systems that are described in the EAST-ADL architectural language. Our aim is to advance both the state of the art and state of practice, and a major emphasis falls on using available techniques in the area of formal verification and model-based testing and in providing a framework relevant to the automotive industry. Given these parameters, we can define our work as being applied, conceptual research, where both descriptive and analytical methods are employed. In Table 4.1, we detail all the activities performed during the research process, and their mapping onto
Chapter 4. Research Methodology

A. Problem Formulation

What is the problem to be solved?
Why is it interesting?
What do we want to achieve?
What are our research goals?

B. Proposed Solution

How do we intend to achieve our goals?

C. Implemented Solution

How do we implement our solution?

D. Validation

How do we ensure applicability and feasibility of our results?
What are our results?
How do we address our research goals?

Figure 4.1: The cycle of the research process.
The research process is achieved through performing a series of activities, such as: (i) defining and redefining problems, (ii) review the relevant literature, (iii) formulate hypotheses and suggesting solutions, (iv) collecting, organizing and evaluating data, (v) making deductions and reaching conclusions, and (vi) testing the conclusions to determine whether they support the hypothesis. In this context, the research methods are defined as concrete techniques that are used to solve a given research problem. They include methods for the collection of data (e.g., data is already available but not sufficient to arrive at the required solution), statistical techniques (e.g., establishing relationships between the data and the unknown), and the evaluation of the accuracy of the results obtained. 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. In computer science, the research process can be described by four major steps [71], as depicted in Figure 4.1.

Our research goal is to investigate how model-checking techniques can be used for the verification of automotive embedded systems that are described in the EAST-ADL architectural language. Our aim is to advance both the state of the art and state of practice, and a major emphasis falls on using available techniques in the area of formal verification and model-based testing and in providing a framework relevant to the automotive industry. Given these parameters, we can define our work as being applied, conceptual research, where both descriptive and analytical methods are employed. In Table 4.1, we detail all the activities performed during the research process, and their mapping onto

Table 4.1: Specific research activities.

<table>
<thead>
<tr>
<th>Specific research activities</th>
<th>Quadrant relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Critical analysis of literature and practice method</strong></td>
<td></td>
</tr>
<tr>
<td>(a) Analyze the state-of-the-art</td>
<td>A</td>
</tr>
<tr>
<td>(b) Analyze the state-of-practice</td>
<td>A</td>
</tr>
<tr>
<td><strong>Formulate research goals</strong></td>
<td>A</td>
</tr>
<tr>
<td><strong>Proof-of-concept research method</strong></td>
<td>A ⇒ B</td>
</tr>
<tr>
<td>(a) Propose analysis framework</td>
<td>A ⇒ B ⇒ C</td>
</tr>
<tr>
<td>(b) Implement analysis framework</td>
<td>A ⇒ B ⇒ C</td>
</tr>
<tr>
<td>(c) Write research papers</td>
<td>A ⇒ B ⇒ C</td>
</tr>
<tr>
<td><strong>Connect theory to practice</strong></td>
<td>A ⇒ B ⇒ C</td>
</tr>
<tr>
<td><strong>Validation method</strong></td>
<td>A ⇒ B ⇒ C ⇒ D</td>
</tr>
</tbody>
</table>

Figure 4.2: Overview of our research process.

The resulting research process is depicted in Figure 4.2. 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 4.1. 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 4.1. 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 (see Table 4.1). 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.
Chapter 5

Thesis Contributions

In this chapter, we present an overview of the entire formal analysis and verification framework and its industrial evaluation. We start by providing a high-level overview of the framework in Section 5.1, and in the later sections we describe the details of the proposed methods, as follows. In Section 5.1 we present a method for defining a formal semantics to the EAST-ADL language, and in Section 5.1 we present our formal verification method for EAST-ADL models extended with formal semantics. In Section 5.5 we augment this framework with model-based testing capabilities based on the findings from the literature review described in Section 5.4. We conclude our work by describing a pruning method for EAST-ADL model, in Section 5.5.

5.1 Overview of the Proposed Framework

Our proposed framework allows for simulation, formal verification and model-based testing of automotive systems modeled in EAST-ADL. The framework is supported by tools like ViTAL [52], which we have developed for automating EAST-ADL model verification, and Farkle [47], a commercial testing tool developed by Alten AB. The workflow consists of the following steps (see Figure 5.1):

   a. Create or import the EAST-ADL model in ViTAL.

   b. Select the verification method:

      (a) Verification with UPPAAL PORT:
Define the behavior of the EAST-ADL model as a network of UPPAAL PORT timed automata;

ii. Analyze the system with the UPPAAL PORT model checker;

iii. Generate abstract test cases with UPPAAL PORT,

iv. Convert the abstract test cases into concrete test cases automatically, by generating Python test scripts;

v. Run the SUT on the concrete test cases, to obtain a pass or fail verdict.

(b) Verification with UPPAAL and UPPAAL SMC:

i. Define the behavior of the EAST-ADL model as a network of UPPAAL timed automata;

ii. Edit the TA associated with the behavior of the corresponding EAST-ADL FunctionPrototypes;

iii. Analyze the system with the UPPAAL model checker;

iv. If UPPAAL-based exhaustive verification does not scale, prune the model based on identified component dependencies and repeat the analysis;

v. Extend the UPPAAL timed automata network with stochastic semantics or resource annotations;

vi. Analyze the system with the UPPAAL SMC model checker.

5.2 Formal Semantics of EAST-ADL Models

In this thesis, we use two different formal notations to describe a system modeled in EAST-ADL, as follows: (i) the UPPAAL PORT timed automata formalism, and (ii) the UPPAAL timed automata formalism.

EAST-ADL 2 UPPAAL PORT TA. As depicted in Section 5.1, we propose a formal semantics for EAST-ADL components in terms of networks of UPPAAL PORT timed automata. In UPPAAL PORT [68], each component is defined by its interface and its timed behavior. We take advantage of the adequacy of such a formal representation for describing component-based models and propose a series of transformations that create a formal model compatible to the input language of UPPAAL PORT. In our approach, the structure of the system, and the interface of the components are modeled in EAST-ADL, whereas the
5.2 Formal Semantics of EAST-ADL Models

In this thesis, we use two different formal notations to describe a system modeled in EAST-ADL, as follows: (i) the UPPAAL PORT timed automata formalism, and (ii) the UPPAAL timed automata formalism.

EAST-ADL 2 UPPAAL PORT TA.

As depicted in Section 5.1, we propose a formal semantics for EAST-ADL components in terms of networks of UPPAAL PORT timed automata. In UPPAAL PORT [68], each component is defined by its interface and its timed behavior. We take advantage of the adequacy of such a formal representation for describing component-based models and propose a series of transformations that create a formal model compatible to the input language of UPPAAL PORT. In our approach, the structure of the system, and the interface of the components are modeled in EAST-ADL, whereas the internal behavior of each component is modeled as an UPPAAL PORT timed automaton [52, 73], respectively.

To translate the architectural model into a formal model, we first map the elements of the EAST-ADL model to an Intermediate Component Model (ICM). The purpose of this transformation is to obtain a mapping from EAST-ADL to the SaveCCM language [41], which is the original modeling language of UPPAAL PORT. The EAST-ADL model is defined as the following tuple:

\[
\text{EAST-ADL Model} \triangleq \langle F_P, DP, Con, TC \rangle, \tag{5.1}
\]

where \(F_P\) is a set of FunctionPrototype, \(DP\) is the set of data ports, defined as the union 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:

\[ ICM \triangleq \langle \text{Comp}, P_{in}, P_{out}, \text{trig}, \text{Connections}, TICM \rangle, \quad (5.2) \]

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

The mapping between the EAST-ADL Model and the ICM is a function:

\[ \pi : EAST-ADL \text{ Model} \rightarrow ICM, \quad (5.3) \]

which maps each \text{FunctionPrototype} 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 \text{FunctionPrototypes}’s timing constraints \( TC \) to ICM’s timing constraints, \( TICM \).

Next, we specify the behavior associated with each EAST-ADL \text{FunctionPrototype} by assigning an UPPAAL PORT TA to the corresponding ICM Comp. The TA encapsulates the internal behavior of the \text{FunctionPrototype}, which in the EAST-ADL language is traditionally defined based on external notations and tools [33]. We define the UPPAAL PORT timed behavior by the following tuple:

\[ B \triangleq \langle L, l_0, l_f, V_D, V_C, r_0, r_f, E, I \rangle, \quad (5.4) \]

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 the sets 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 actions, guards, and sets of clocks to be reset, and \( I \) assigns clock invariants to locations.

In order to integrate the two models, ICM and the set of TA behaviors, we implement another transformation that extends the UPPAAL PORT timed behavior as follows:

\[ TA \triangleq \langle L \cup \{l_{\perp}\}, l_0, l_f, V_C, V_D, r_0, r_f, E, I \rangle, \quad (5.5) \]

where the set of TA locations \( L \) is extended with the idle location \( l_{\perp} \), representing the location of the TA that corresponds to a \text{FunctionPrototype} not being active. Originally, no edges in \( E \) are leading to \( l_0 \) or from \( l_f \), respectively. For this, we define the action \text{Read}(P_{in}) that reads the values of input port variables and \text{Write}(P_{out}) that writes the variables onto output ports. The other
5.2 Formal Semantics of EAST-ADL Models

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. This, in turn, enables the formal verification of the models (originally described in EAST-ADL) with the UPPAAL PORT model checker and provides a verification framework that allows for modeling, simulation, and formal verification of functional and timing system requirements, based on component-based model-checking techniques.

EAST-ADL 2 UPPAAL TA. As mentioned in Section 5.1, we also propose a model-to-model transformation from EAST-ADL to networks of UPPAAL TA. To achieve this, we develop an automatic transformation that takes the EAST-ADL Model tuple of Equation 5.1, and maps it to a parallel composition of TA, where an UPPAAL TA is defined according to Equation 2.3.

\[\pi : EAST - ADL\ Model \rightarrow TA\] (5.6)

The transformation is a one-to-one mapping function:

- Each \(F_P\) is defined in terms of a network of two TA, a TA corresponding to the \(F_P\)’s interface, and a TA encoding its (internal) behavior, as shown in Figure 5.2. To preserve the “read-execute-write” semantics of EAST-ADL, the Interface TA (see Figure 5.2 (a)) has four locations: (i) Idle, (ii) a Read location that allows the update of the variables according to the values at the input ports, independent of other computations, (iii) an Exec location that triggers the Behavior TA (see Figure 5.2 (b)) that models the desired behavior of \(F_P\), and (iv) a Write location that allows the update of the output ports according to the values of the computed internal variables, respectively, independent of other computations.
Chapter 5. Thesis Contributions

- Each input and output port $DP$ is mapped to a global variable in the TA network, respectively.

- Each connector $Con$ from output port $Port_{out1}$ of $F_{P1}$ to input port $Port_{in2}$ of $F_{P2}$ is transformed into an assignment $Port_{in2} := Port_{out1}$ along the edge from $Idle$ to $Read$;

- The triggering of each interface TA is based on the triggering associated to the EAST-ADL $F_p$. Concretely, this creates two possible instantiations of the Interface TA: (i) for timed-triggered $F_p$, the transformation produces a local clock, plus invariants and guards on TA, and (ii) for event-triggered $F_p$ the transformation produces a set of dedicated variables that need to be constantly updated and reset, respectively.

- Other timing annotations $TC$, e.g., the execution time, can be included in the timing behavior of the TA model.

Once we obtain the network of TA corresponding to the EAST-ADL model, one can manually edit the Behavior TA to match the desired behavior of the corresponding FunctionPrototype. This transformation enables one to analyze the formal model with the UPPAAL model checker. This formal model can also be manually extended with stochastic semantics, and thus be analyzed with the UPPAAL SMC statistical model checker to estimate probabilities and probability distributions over time with given confidence levels. Similarly, the formal model can be manually extended with resource annotations (based on the information provided in the architectural model) creating a network of priced timed automata that can be analyzed with UPPAAL SMC to provide information about the resource usage of the system. In the next section, we show the analysis of theses models with UPPAAL PORT, UPPAAL, and UPPAAL SMC, and we show how such an analysis provides valuable feedback on the system’s correctness prior to the actual implementation.

5.3 Formal Analysis of EAST-ADL Models

UPPAAL PORT [68] supports simulation and model checking of component-based systems, without the usual flattening of the underlying TA network. The tool uses Partial Order Reduction Techniques (PORT) to improve the efficiency of verification, and local time semantics [21] to increase independence, being suited for model checking “read-execute-write” component models, like the
ones described in EAST-ADL. Providing formal semantics to the EAST-ADL architectural model in terms of UPPAAL PORT TA enables the formal verification of the models (originally described in EAST-ADL) with the UPPAAL PORT model checker, which bounds the state space during verification, and possibly overcomes the potential state-space explosion problem.

Similar to the UPPAAL PORT approach, the definition of EAST-ADL component semantics in terms of regular UPPAAL TA enables employing UPPAAL rather than UPPAAL PORT for simulation and formal verification of EAST-ADL models. On the one hand, UPPAAL TA, as compared to UPPAAL PORT TA allow the use of synchronization channels between automata and other constructs that make modeling more flexible. On the other hand, without the partial order reduction and the local time semantics, the verification with UPPAAL might not scale due to state-space explosion. In this case, we can apply a pruning algorithm that identifies the dependencies within the model and reduces the latter to a model that contains only the relevant components. This pruning ensures exhaustive verification on a downscaled, equivalent model.

The EAST-ADL 2 UPPAAL TA transformation also enables the analysis of the formal model with UPPAAL SMC, thus providing probabilistic analysis of resource-usage estimations for EAST-ADL. For this, we manually extend the network of UPPAAL TA with stochastic semantics, and use UPPAAL SMC for statistical model checking. UPPAAL SMC verifies qualitative properties in terms of probabilities and cost. The UPPAAL TA network is then manually extended with resource annotations (based on the information provided in the architectural model) generating semantic descriptions as networks of priced timed automata. We analyze this model statistically, to compute worst case or average resource usage of EAST-ADL components (quantitative analysis), as well as check whether accumulated resource usage does not exceed a certain upper bound that can be considered as the provided resource amount (feasibility analysis). We show how bounded analysis techniques, like statistical model checking, can be applied on high-level design artifacts.

5.4 Tool supported Model-based Testing: Literature Review

We have performed an extensive review to provide an overview of the model-based testing approaches for requirements-based designs available in the technical literature. To achieve this, we propose a taxonomy that allows us to classify such relevant testing frameworks. In 2012, Utting et al. [112] devised a taxonomy that identified different dimensions of model-based testing. From
this taxonomy, we include four of the proposed dimensions into our proposed taxonomy, namely the model notations, the test selection criteria, the test generation method, and the test execution. We then extend our selection with two new dimension:

- The test artifact represented by the model, and
- 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. This extension is justified by the fact that it is important to assess the capability of a testing framework to test extra functional behaviors, or be able to generate executable test cases that can test the actual implementation.

Our end goal is to supply 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. Due to the large number of models proposed by researchers, we focus only on model-based testing techniques supported by integrated tools, which handle modeling, test-case generation, test-case selection and possibly execution. Our inclusion criteria narrows the review to 33 tools, which are described and classified according to the taxonomy. This classification helps to identify differences and common characteristics among existing techniques, but also possible gaps in the existing techniques. For instance, we observe that most test-case generation tools use transition-based notations and model-checking techniques to generate test cases. Our results also show that most tools focus on generating test cases for functional properties, while only a fraction focus on extra-functional properties (i.e., only three tools deal with extra-functional artifacts) and none of the considered model-based testing tools deal directly with architectural artifacts. To provide further evidence of the inner workings of different model-based testing tools, we select a set of representative tools and we apply them on a simple yet illustrative Coffee/Tea Vending Machine example (an extension of the Coffee Vending Machine used to exemplify the TA formalism), to show the differences in modeling notations, test case generation methods, and the produced test cases [92].

5.5 Model-based Testing of Automotive Systems

Our findings, stemmed from investigating the model-based testing tools of requirement-based designs, have motivated the extension of our formal frame-
work with model-based testing capabilities. To achieve this, we first provide executable semantics for the EAST-ADL language with UPPAAL PORT TA semantics, and then we employ the same verification framework to generate test cases. Moreover, we need to convert the abstract test cases into executable test scripts that can be run against the system implementation to provide test verdicts. This last step is also a way to assess the feasibility of our model-generated abstract test cases, in the sense of evaluating their appropriateness for obtaining rich enough executable test cases that the SUT can be run on, and the execution terminates with fail or pass verdicts.

**Implementation guidelines.** The TA behavior of an EAST-ADL FunctionPrototype can be non-deterministic, so, in order to obtain an implementation of the EAST-ADL model, we need to determinize the model. For this, we define an executable (deterministic) semantics of the UPPAAL PORT TA in a similar manner as Amnell et al.[14] have done 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 [14], we resolve non-determinism, as follows:

- Action non-determinism is resolved by implementing a function that assigns unique priorities to each edge in the UPPAAL PORT TA to establish 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 [119], 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.

These guidelines provide 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 the preservation of the safety properties of the EAST-ADL behavioral TA model. We approach the problem of code implementation as a mapping activity between the EAST-ADL system model extended with TA semantics and the C code [91].
Test-case generation. In order to verify that the system’s implementation conforms with its requirements, we extend our verification framework with test-case generation capabilities. For this, we employ UPPAAL PORT to automate the abstract test-case generation, by exploiting the ability of the model checker to automatically 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 TCTL reachability properties. The model checker returns a witness trace, which is a sequence of states and transitions of the model, representing our abstract test case for a particular test goal. In the 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. In the underlying semantic representation, the transitions can be either delay, internal, read, or write transitions. This abstract test-case generation methodology is applied to both functional and timing requirements.

Test-case conversion and execution. In order to execute the test cases, i.e., abstract test cases (ATC), we need to convert them into executable ones, that is, test scripts (in our case implemented in Python). The pseudo algorithm for test-script generation and execution against the SUT is shown in Figure 5.3. To achieve the test-case conversion, the ATC are read (line 1 in Figure 5.3) and parsed (line 2 in Figure 5.3) to identify all the states and transitions, along with the value of variables at each state (line 3 in Figure 5.3). This information is stored in the Python script as follows: (i) the various values for each variable are stored in a dedicated set of arrays (line 6-9 in Figure 5.3), and (ii) the sequence of states are stored in another array (line 11-13 in Figure 5.3). This information is used (as a signal) by the script to trigger the execution of the system in order to verify whether the same inputs to the SUT provide the same set of states and transitions and in the exact order specified by the ATC during execution (lines 15-16 in Figure 5.3).

This algorithm also includes the execution of the test scripts, detailed for both functional and timing test scripts. The Python test script triggers the execution of the system by sending a signal to the SUT, based on the initial values of variables parsed from the ATC (line 17 in Figure 5.3). The actual set of states and transitions that are executed at runtime and the value of variables at each state are collected and logged, and are sent to the test script using the signals communication mechanism (line 20 in Figure 5.3). The test log is evaluated to decide if there is any deviation from the expected behavior as specified in
5.5 Model-based Testing of Automotive Systems

Read the ATC.

Parse the ATC.

Identify different variables, states, and existence of timing requirements in the ATC.

For each variable in the ATC:

Create an array containing its values at each state in the order they appear in ATC.

// e.g., ABSFL_v = [6, 7, 9]

Create an array to keep order of states as they appear in the ATC.
// i.e., ID of each state is stored in the order
// e.g., ABSFL_state = [1, 2, 3]

The initial values for each variable are stored as part of a signal to be sent to the target.
Send the signal to the target.

// the system is triggered by the signal
Receive the signal containing the log info from the target.

// functional requirement
TestResult = Pass

For each log record in the signal:

For each variable in the log record

Add an ‘assert’ statement checking its value against the respective array index containing the expected values of the variable (including the states order array).

If there is a mismatch then:

TestResult = Fail.

Print previous state, current state, variable name, actual value, expected value, state timestamp.

// timing requirement

If a requirement on end-to-end response time is identified in the ATC:

Get timestamp of the last state in the last component.
Get timestamp of the first state in the first component.
Calculate the time by reducing the above values.
Compare the result against the requirement value.

If they do not match then:

TestResult = Fail.

Print ‘Timing violation’, expected response time, actual response time.

Print TestResult

Figure 5.3: Test-Script Generation Pseudo Algorithm.

the ATC. If any discrepancy between the actual and the expected orders of the traversed states or the produced variables values is found, the test result will be fail, otherwise a pass verdict is issued (line 22-33 in Figure 5.3). In other
words, during the test execution, the role of the Python script is to check that
given the inputs, the exact same order of states as in the ATC is also observed
at runtime. Such a conformance checking is made possible in our approach by
implementing the code based on a switch-case structure among different states.

In order to be able to test timing requirements, a timestamp is included in
the log record each time a state change occurs in the system. By consulting
these timestamps it becomes possible to determine at which time point a state
has been visited. Moreover, from these timestamps it can also be determined
how long it has taken to go from one state to another (line 35-43 in Figure
5.3). Finally, the test verdict is displayed (line 45 in Figure 5.3). However,
the verdicts in case of testing timing behavior can be quite inaccurate, as they
depend on the implementation platform.

5.6 Pruning EAST-ADL Models by Dependency
Analysis

Attempting to verify complex industrial systems through exhaustive verifi-
cation might not scale as desired. Even in cases when the state-space explosion
phenomenon does not occur, the actual exhaustive verification might be time
and memory costly anyway. We attempt to overcome this problem and we pro-
pose a pruning method tailored for EAST-ADL architectural models. We base
our work on the dependency analysis method proposed by Stafford et al. [110].

The analysis starts by identifying "links", dependency relationships within
the architecture, which directly connect elements of the model. Direct depen-
dencies can be related either to the structure or to the behavior of the model.
These links are annotated in a matrix and a straightforward algorithm is used
to compute the dependency chains in the model. Once the dependency matrix
has been computed, it can be used to provide input on the dependencies that
are related, either directly or indirectly, to a particular port. Such dependency
chains can be, for example affected-by chains, consisting of the set of compo-
nents that can potentially affect the behavior of another component. The end
result of this work is a set of dependency chains that can be used in the analysis
of the system.

In our work, the UPPAAL TA model is parsed in order to generate the de-
pendency matrix. The dependencies are stored in an $m \times m$ matrix, where
$m$ represents the number of ports in the EAST-ADL model. These ports are
mapped to global variables in the timed automata model during the transfor-
5.6 Pruning EAST-ADL Models by Dependency

Analysis

mation, while the actual dependencies are considered to be any assignment function of these variables. For this, we look at all the transitions in all the timed automata in the model. Each left-hand side variable from an assignment depends on all the variables in the right-hand side of the assignment, and all the variables in the corresponding guards. If these elements are port variables (and not local variables of the TA), then this dependency is marked in the matrix. The resulting matrix can be interpreted as follows: the elements of the rows represent the starting point of the dependency, whereas the elements of the column represent the ending point of the dependency link. Based on the dependency matrix, we determine the dependency chain for a particular property. Given an EAST-ADL component, we want to isolate the chains consisting of the set of components that can affect the execution of the chosen component. Concretely, we want to list all the elements whose execution precedes and impacts the execution of our given element by identifying all the direct and indirect links. For this, we examine the requirement that we want to analyze and for which we intend to prune the model, by identifying the relevant EAST-ADL ports and the corresponding TA variables. For each port variable, we parse the dependency matrix to identify all the dependencies. The corresponding port variables of each dependency are added to the list of variables to be examined, until there are no more dependencies to examine. The result of this algorithm is a list of dependencies that includes the list of variables associated with EAST-ADL ports that are in the dependency chain.

Once the dependency chain for a particular property is generated, we prune the network of TA to include only the elements of the dependency chain. For each FunctionPrototype in the original EAST-ADL model, the pruning is done based on the following 3 possible scenarios:

- **Scenario 1**: If no port variables are listed in the Passed_Ports list, then the corresponding Interface and Behavior TA are removed from the network;

- **Scenario 2**: If only a subset of ports are listed in the Passed_Ports list, then the Interface and Behavior TA are updated to include only the information about these ports;

- **Scenario 3**: If all ports are listed in the Passed_Ports list, then the model remains unchanged.

With this pruning algorithm, we can ensure that the model contains only the relevant information of the original EAST-ADL model, with respect to the verification of a certain property. Moreover, pruning the model based on the identified dependency chains ensures that relevant elements will not be removed
from the model. Our work also includes a proof of the correctness of the pruning algorithm, which is presented in Paper F [90].

5.7 Tool Support: ViTAL

In this thesis, we develop ViTAL (Verification Tool for EAST-ADL Models using UPPAAL/UPPAAL PORT) as the tool support for our framework presented in Section 5.1. ViTAL integrates architectural languages and verification techniques, and hence it provides 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 5.4. 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 FunctionPrototype. The mapping between these two models is used by the analysis and test-case generation interface of UPPAAL PORT, which relies on a client-server architecture. The test-case generation interface communicates with Farkle [47], which is a commercial tool for test-case conversion (to Python test scripts) and execution. For details on Farkle, we refer to reader to the literature [47, 91].

![ViTAL tool diagram](image)

Figure 5.4: The ViTAL tool.

In a similar manner, a mapping of the EAST-ADL model to a network of UPPAAL TA 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. For this model we also propose a pruning algorithm that
reads the network of UPPAAL TA and, for a given requirement, provides an equivalent pruned model. This algorithm has not yet been implemented in the ViTAL tool, thus it has been marked with a dotted line in Figure 5.4. Using these two verification interfaces, it is possible to validate the behavior and timing of EAST-ADL system models prior to implementation.

5.8 Validation on the Brake-by-Wire Use Case

Our methodology introduced in Section 5.1, supported by the ViTAL-Farkle toolchain has been applied on the Brake-by-Wire (BBW) case study for validation purposes. In this section, we show how the BBW system can be simulated and formally verified against different functional and timing properties with UPPAAL PORT [52, 73]. Next, we show 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 on which the actual code is finally run. Our work is an attempt to exercise the feasibility of test case generation from EAST-ADL models [91]. The formal verification and model-based testing methodology has shown encouraging results when applied on the BBW system, which is a system prototype developed by AB Volvo (see our published work [73, 89, 90, 91]).

We also show how the BBW system can be simulated and formally verified against different functional and extra-functional properties with UPPAAL [89] and UPPAAL SMC [89]. UPPAAL SMC allows one to apply the propose methodology to analyze the resource consumption of the system. Finally, we show how such a model can be pruned to provide a significant state space reduction during verification, which we actually show on the chosen use case.

**UPPAAL PORT formal semantics.** Figure 5.5 depicts a possible behavioral model associated with the pABS_FL FunctionPrototype of Figure 2.3, in terms of UPPAAL PORT TA. The behavior of the TA model is described as follows. First, the speed of the vehicle ($v$) is evaluated: if the vehicle has no speed then no brake force is applied ($torqueABS = 0$), 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 $torqueABS = 0$), otherwise the desired braking torque $wheelABS$ is sent to the corresponding actuator ($torqueABS = wheelABS$).
For the \textit{pABS\_FL FunctionPrototype} and the associated TA behavior (presented in Figure 5.5), 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 5.6) and (ii) the behavior of the \textit{FunctionPrototype} extended with the \textit{idle} location and the \textit{read} and \textit{write} actions (see Figure 5.7). The \textit{idle} state (line 2) is also the initial state of the system (line 3), and the resulting possible transitions of the TA are the \textit{read} action that is represented by the transition from \textit{Idle} to \textit{Entry} (line 4), the internal transitions of the TA model (lines 5-15), and the \textit{write} action that is represented by the transition from \textit{Exit} to \textit{Idle} (line 16).

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{fig5.5.png}
\caption{The TA model associated with the \textit{pABS\_FL FunctionPrototype}.}
\end{figure}

\begin{verbatim}
<MODEL type="uppaal:declarations">
int wheelABS, torqueABS, v, w;
int R=1;
</MODEL><MODEL type="uppaal:behaviour">
state idle, Entry, CalcSlipRate, Exit;
init idle;
trans idle->Entry { guard false; },
Entry->Exit {guard v==0; assign torqueABS=0;},
Entry->CalcSlipRate {guard v>0;},
CalcSlipRate->Exit {guard v>=5*(v-w*R); assign torqueABS=wheelABS;},
CalcSlipRate->Exit {guard v<5*(v-w*R); assign torqueABS=0;},
Exit->idle {guard false;};
</MODEL></BEHAVIOIR>

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{fig5.6.png}
\caption{Mapping between UPPAAL PORT TA local variables and the EAST-ADL ports for the \textit{pABS\_FL}.}
\end{figure}

\textbf{Formal verification of the EAST-ADL model extended with UPPAAL PORT semantics.} With UPPAAL PORT, we can symbolically simulate, as well as exhaustively check the model to verify if it meets its requirements. For requirement \(R_1\) introduced in Section 2.1, the CTL query is as follows:

\[
\text{A}[] \text{ABS.v} > 0 \text{ and } \text{ABS.v} < 5(\text{ABS.v} - \text{ABS.w} \times \text{ABS.R}) \implies \text{WheelActuator.NoBrake}
\]

The verification results are presented in more detail in our work [52, 73].
5.8 Validation on the Brake-by-Wire Use Case

Implementation based on the EAST-ADL model extended with UPPAAL PORT semantics. Based on the implementation guidelines, we have implemented the code for the BBW system. A section of the code, depicting the functionality of the ABS component is shown in Figure 5.8.

Each location in the TA model depicted in Figure 5.5 represents a possible value of the variable state, which is initially set to Entry. While state is different from Exit, the code implements all the possible computations of the TA, e.g., if $v = 0$ then $\text{torqueABS}$ is set to zero. We note here that time is not considered in this implementation, so all transitions are taken instantly.

Abstract test-case generation. We employ UPPAAL PORT to generate abstract test cases from the abstract model previously constructed. The model checker takes as input the formal model together with the test goal specified as the CTL reachability property. For requirement $R_1$ introduced in Section 2.1, the CTL query is expressed as follows:

$$E < > \text{ABS}.v > 0 \text{ and ABS}.v < 5 \times (\text{ABS}.v - \text{ABS}.w \times \text{ABS}.R) \text{ and WheelActuator.NoBrake}$$

The UPPAAL PORT model checker automatically generates the witness trace presented in Figure 5.9, which represents the execution of the $pABS_FL$ FunctionPrototype. Initially, the TA is in location idle and all variables are zero. The first transition to state Entry is a read transition, where the latest variable values of $w$, wheelABS, and $v$ are read. Since $v > 0$, the TA moves to the CalcSliprate location. On the transition to Exit, the $\text{torqueABS}$ variable is
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 5.8: Implementation of the ABS component

set to zero, and after the write transition, the TA returns to the idle location.

Test-case conversion and execution. From the abstract test case shown in
Figure 5.9, the input variable values determining transitions in the TA model
are automatically identified, and a Python test script is generated. An excerpt
of the generated script is shown in Figure 5.10. Lines 6-10 in the script set the
initial values as the content of the input signal that is then sent to the target (line
11). The expected variable values, as well as the expected order of visited states
are defined in lines 13-18, according to the principles described in Section
5.5. The log information sent back to the host, from the target, in the form
of a signal is then received (line 20). Again, following the aforementioned
principles, a set of assertion statements for checking the returned values versus
the expected values are also generated as the body of a loop, depicted in lines
23-39 of the script.

When the test script is executed, an input signal is sent to the target. Upon
receiving the signal, the process starts executing. The different states that a
5.8 Validation on the Brake-by-Wire Use Case

Figure 5.8: Implementation of the ABS component

```c
State:(ABSFL.idle)
  ABSFL.w=0 ABSFL.wheelABS=0 ABSFL.torqueABS=-1
  ABSFL.v=0 ABSFL.R=1/2
Transitions: ABSFL.idle->ABSFL.Entry { w:= 8, wheelABS:= 1,
  v:= 12 }
State:(ABSFL.Entry)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1
  ABSFL.v=12 ABSFL.R=1/2
Transitions: ABSFL.Entry->ABSFL.CalcSlipRate { v> 0 }
State:(ABSFL.CalcSlipRate)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=-1
  ABSFL.v=12 ABSFL.R=1/2
Transitions: ABSFL.CalcSlipRate->ABSFL.Exit
  { v< 5* (v- w* R), torqueABS:= 0 }
State: (ABSFL.Exit)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0
  ABSFL.v=12 ABSFL.R=1/2
Transitions: ABSFL.Exit->ABSFL.idle { }
State: (ABSFL.idle)
  ABSFL.w=8 ABSFL.wheelABS=1 ABSFL.torqueABS=0
  ABSFL.v=12 ABSFL.R=1/2
```

Figure 5.9: Abstract Test Case

The process enters are tracked and logged at runtime and during the execution of the code. This information is sent back to Farkle, where the order of the states at runtime and the variable values after each state change is checked to match the specification in the abstract test case generated from the TA models. The Python script of Figure 5.10 delivers a “pass” verdict on the implementation of Figure 5.8.

**UPPAAL-based formal semantics.** We have applied our transformation from EAST-ADL to a network of UPPAAL TA on the BBW system. The result is a network of 50 TA, where each of the 25 `FunctionPrototype` of Figure 2.3 is transformed into a parallel composition of two synchronized TA, respectively. In Figures 5.11, we exemplify the transformation of the `pABS_FL FunctionPrototype` as follows: Figure 5.11 (a) presents the interface of the time-triggered `pABS_FL FunctionPrototype` automatically generated from the EAST-ADL model, and Figure 5.11 (b) presents the behavior of the `pABS_FL FunctionPrototype` obtained after manually editing the dedicated TA template (see Figure 5.2(b)).

**Formal verification with UPPAAL and UPPAAL SMC.** With UPPAAL, we have simulated and we have attempted to verify the previously described net-
work of TA. However, the size of the model leads to state space explosion. On a computer with 1.8 Ghz Intel processor and 8GB memory, the verifier explored only 10 962 377 states before running out of memory. This is not surprising, since the BBW system is subject to an enormous state-space explosion due to the large number of TA in the network, each with its clock and its set of variables created based on the ports of the corresponding FunctionPrototype.

Since UPPAAL SMC works on stochastic models, we manually add proba-
5.8 Validation on the Brake-by-Wire Use Case

bilistic extensions, to the four-wheels BBW model that contains the timed behavior. Figures 5.12 (a) and 5.12 (b) show exponential rates added to locations Idle and Exec of one Encoder component of Figure 2.3. The rate of 1 means that the component may potentially stay forever in the location, but it will stay there for 1 time unit on average, which is consistent with the timed behavior. Further, we are interested in the latency between pressing the pedal and applying the brakes, hence we added a monitoring stop-watch automaton shown in Figure 9.11c. The monitoring automaton has a stop-watch \( L \) that is stopped originally in location Wait by specifying that the derivative is zero: \( L' = 0 \). The stop-watch is started when synchronization \( pBrakePedalSensor_{beh\_start} \) is received (the derivative \( L' = 1 \) is implicit in timed automata). The stop-watch is stopped again when any of the wheels receive the synchronization breaking signal, like \( pHW_{Brake\_FL\_beh\_start} \) or \( pHW_{Brake\_FR\_beh\_start} \) (the synchronizations are on different edges that are drawn on top of each other to minimize cluttering). The latency can be estimated by the following query: \( Pr[bm.L <= 1000](<> bm.Done) \) that asks what is the probability that the
brake monitor process $bm$ will end up in location $Done$ in terms of the stopwatch $L$ value. The result is shown in Figure 5.12 (d). The average latency is 5 time units but it tends to be high even though our added stochastic delay assumptions are decreasing towards infinity, which is a worrying behavior. We note here that this behavior seems to be strictly limited by 6 time units and no simulation has been observed greater or equal than 6 time units, which is on the other hand surprising, as the model contains components with unlimited delays.

Being equipped with the formal model of the BBW system, we next focus on the hardware elements of the platform, and their available resources, as they are represented in EAST-ADL. For the BBW system, we are interested in the $pECU_Central$ hardware component dedicated to the following pedal computational elements: $pBrakePedalLDM$, $pBrakeTorqueMap$, $pGlobalBrakeController$, and $pVehicleSpeedEstimator$, which are allocated to this ECU. Energy is consumed during the execution of the ECU. Similarly, memory is allocated before the component is executed, and it is deallocated at the end of the execution. Figure 10.7 depicts the simulation of the energy consumption of the four components allocated on the $pECU_Central$ hardware component. Concretely, the plot shows one stochastic simulation carried out for 50 time units, obtained by using the following query on the model: simulate $\{energy, memory, t\mid L < 50\}$. We also evaluate the maximum expected value for the energy. For this, we simulate the system over 2000 runs, trying to maximize the energy consumption, with the query $E[\{max\ energy\mid t < 50, 2000\}]$. The mean value provided by UPPAAL SMC for the maximum consumption is around 89.

Pruning EAST-ADL models extended with UPPAAL semantics. Assuming the formal model, we intend to verify the end-to-end deadline requirement $R_7$ (see Table 2.1). For this, we implement the monitor shown in Figure 5.14. The end-to-end requirement is formalized as a TCTL property, as follows:

$$A[Monitor.End \implies Monitor.x < 2000]$$

(5.7)

Due to the state-space explosion problem, this property cannot be verified. For the verification to scale on a 1.8 GHz Intel Core i5 processor with 8 GB of memory, all variables except the clocks have been marked as "meta", meaning that they are stored in the state vector, but are not considered part of the state. The TCTL property is satisfied on the formal model and the UPPAAL model checker explores 5060336 states during verification.
5.8 Validation on the Brake-by-Wire Use Case

consumption of the four components allocated on the pECU_Central hardware component. Concretely, the plot shows one stochastic simulation carried out for 50 time units, obtained by using the following query on the model: \( \text{simulate} [1 <= 50] \{\text{energy, memory, } t\} \).

We also evaluate the maximum expected value for the energy. For this, we simulate the system over 2000 runs, trying to maximize the energy consumption, with the query \( E[t <= 50, 2000](\max : \text{energy}) \). The mean value provided by UPPAAL SMC for the maximum consumption is around 89.

**Pruning EAST-ADL models extended with UPPAAL semantics.** Assuming the formal model, we intend to verify the end-to-end deadline requirement \( R_7 \) (see Table 2.1). For this, we implement the monitor shown in Figure 5.14. The end-to-end requirement is formalized as a TCTL property, as follows:

\[
A[] \text{Monitor.End imply Monitor.x <= 2000} \tag{5.7}
\]

Due to the state-space explosion problem, this property cannot be verified. For the verification to scale on a 1.8 GHz Intel Core i5 processor with 8 GB of memory, all variables except the clocks have been marked as “meta”, meaning that they are stored in the state vector, but are not considered part of the state. The TCTL property is satisfied on the formal model and the UPPAAL model checker explores 5060336 states during verification.
To further reduce the state space, we apply the dependency method introduced in Section 5.5. The result is a 50x50 matrix, where the size of the matrix is dictated by the number of ports (that is, there are in total 50 input and output ports in the EAST-ADL model). The dependency matrix for the Brake-by-Wire system is presented in Paper F [90].

In order to prune the model, we apply the pruning method introduced in Section 5.5. Only the elements of this dependency chain are needed during analysis, while the other elements can be pruned away. As a result of the dependency analysis, six FunctionPrototypes can be removed from the formal model (i.e., \( p_{ABS\_FR}, p_{ABS\_RL}, p_{ABS\_RR}, p_{LDM\_Brake\_FR}, p_{LDM\_Brake\_RL}, \) and \( p_{LDM\_Brake\_RR} \)). Also, for the \( p_{GlobalBrakeController} \), only the information from two input ports (i.e., \( DriverReqTorqueIn \) and \( WheelSpeed\_FLIn \)) is needed to compute the corresponding output port (i.e., \( TorqRef\_FLOut \)).

The network of TA, representing the BBW formal model, has been pruned according to the dependency links. On this pruned model, we have verified the TCTL property describing requirement \( R_7 \), which is satisfied. During the verification, 66740 states have been explored, representing around 1.32% of the state-space explored for the original model.

5.9 Research Goals Revisited

In this section we present the contribution of each of the six papers and we show how they address the research goals formulated in Section 3.2.

5.9.1 Paper A

In Paper A [73], we present the use of formal modeling and verification techniques at an early development stage of automotive embedded systems that are...
To further reduce the state space, we apply the dependency method introduced in Section 5.5. The result is a 50x50 matrix, where the size of the matrix is dictated by the number of ports (that is, there are in total 50 input and output ports in the EAST-ADL model). The dependency matrix for the Brake-by-Wire system is presented in Paper F [90].

In order to prune the model, we apply the pruning method introduced in Section 5.5. Only the elements of this dependency chain are needed during analysis, while the other elements can be pruned away. As a result of the dependency analysis, six FunctionPrototypes can be removed from the formal model (i.e., pABS FR, pABS RL, pABS RR, pLDM Brake FR, pLDM Brake RL, and pLDM Brake RR). Also, for the pGlobalBrakeController, only the information from two input ports (i.e, DriverReqTorqueIn and WheelSpeed FLIn) is needed to compute the corresponding output port (i.e., TorqRef FLOut).

The network of TA, representing the BBW formal model, has been pruned according to the dependency links. On this pruned model, we have verified the TCTL property describing requirement R7, which is satisfied. During the verification, 66740 states have been explored, representing around 1.32% of the state-space explored for the original model.

5.9 Research Goals Revisited

In this section we present the contribution of each of the six papers and we show how they address the research goals formulated in Section 3.2.

### 5.9.1 Paper A

In Paper A [73], we present the use of formal modeling and verification techniques at an early development stage of automotive embedded systems that are originally described in EAST-ADL. We employ the UPPAAL PORT timed automata formalism to capture the behavior of each FunctionPrototype in the EAST-ADL model, while the architectural model defines the structure of the system (RG 1). The proposed verification framework allows for modeling, simulation, and formal verification of functional and timing requirements of the system based on model-checking techniques (RG 2). The independence introduced by the read-execute-write semantics that characterizes the EAST-ADL functional modeling, is exploited by UPPAAL PORT in order to reduce time and space requirements for model checking. Our technique improves the behavior modeling, verification, and analysis capabilities of EAST-ADL, and the result shows the applicability of model checking on automotive complex embedded systems (RG 6). This framework is later extended in Paper E to allow for abstract test-case generation based on the EAST-ADL models extended with timed automata semantics (RG 4).

### 5.9.2 Paper B

In Paper B [89], 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 (RG 2) and (iii) the statistical model checking of the architectural model with UPPAAL SMC (RG 2). We show how the analysis techniques underlying the tools complement each other, by applying the transformations and tools to analyze the BBW industrial system. The actual contribution of this paper consists in introducing two new trans-
formations, one from EAST-ADL models to Simulink models, and one from EAST-ADL models to UPPAAL models (RG 1), together with the application of simulation, model checking and statistical model checking on an industrial architectural model (RG 6).

5.9.3 Paper C

In Paper C [88], we start with the transformation from EAST-ADL models to UPPAAL models introduced in Paper B, and we show how efficient verification techniques, like statistical model checking, can be applied to provide early information on the resource consumption of an automotive embedded system. To achieve this, we extend the network of timed automata that encodes the behavior of the FunctionPrototypes, with resource annotations based on the information provided in the architectural model, thus creating networks of priced timed automata, which are extensions of timed automata with costs. Consequently, we can employ UPPAAL SMC to analyze the resource usage of automotive systems described in EAST-ADL (RG 2). The results of the analysis, applied on the Brake-by-Wire industrial prototype, provide valuable information on the system’s resource-usage prior to the actual implementation, such as the feasibility and worst-case resource consumption of the embedded components (RG 6).

5.9.4 Paper D

In Paper D [92], 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 an 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 to applicability of model-based testing (RG 3).
5.9.5 Paper E

In Paper E, we introduce a methodology for model-based testing against functional and timing requirements of embedded systems, starting from EAST-ADL architectural models (RG 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 SUT on the corresponding executable test cases, in an attempt to obtain pass or fail verdicts (RG 6). If the endeavor succeeds, the testing effort could be reduced by the automatic provision of valid test cases, which consequently leads to reducing testing time. Applying the methodology on the BBW system shows promising results.

5.9.6 Paper F

In Paper F [90], we propose a method for pruning EAST-ADL models, based on the dependency chain identification technique developed by Stafford et al [110]. Our method subsumes two steps: (i) computing the dependency matrix from the semantic representation of the EAST-ADL model as a network of timed automata (presented in Paper B), and parsing it into dependency chains, and (ii) pruning the architectural model based on the obtained dependency chains, with respect to a given requirement, such as the end-to-end delay. We propose three algorithms to automate these steps, that is, one for creating the dependency matrix, one for identifying the dependency chains in the EAST-ADL model, and one for reducing the latter by eliminating the components and connections that do not contribute to executions related to the property under consideration. The types of chains that are identified by the method are intra- and inter-component chains, which reveal the dependencies between the input and output ports of a component alone, and the dependencies between the ports of different components, respectively. (RG 5). The methodology is illustrated on the Brake-by-Wire industrial system (RG 6).
Chapter 6
Related Work
Software architecture has been explored in a considerable amount of work \[29, 46, 58, 108\] in the last two decades, leading to contributions in software design \[6, 116\], software reuse \[45, 59, 60\], software evolution \[16, 35\], analysis \[8, 53, 62, 121\], and testing \[26, 27\]. Analyzing software architectures is a challenging task that has received significant attention from the research community during the past few years. Zhang et al. \[121\] have compared and classified over 16 approaches that analyze software architectures using model-checking techniques, which shows that this research topic is of growing interest. For example, Allen \[9\] applies model-checking techniques directly at the (software) architecture level, work that is later extended to also address the problem of specifying and analyzing dynamic architectures \[7\], whereas Burmester \[40\] proposes an approach for analyzing safety-critical, real-time systems. Compared to these contributions, which use a formal description and could support various kinds of formal verification, our framework is not limited to the use of exhaustive model checking and supports the use of model-based testing and statistical model checking. Thus our framework can be applied to a larger class of behaviors (e.g., infinite state behaviors).

In the automotive domain, two architectural languages have received special attention, E\textsuperscript{AST}-ADL \[33\] and AADL (Architecture Analysis and Design Language) \[54\]. Several studies have been carried out towards the analysis of AADLs models enriched with its behavioral annex \[25, 32\]. Berthomieu et al. \[25\] use the Fiacre language for analyzing the functional behavior with the TINA tool. Our framework carries along the same goal: to provide a formal semantics to the architectural language, which in this case is AADL and not...
Chapter 6

Related Work

Software architecture has been explored in a considerable amount of work [29, 46, 58, 108] in the last two decades, leading to contributions in software design [6, 116], software reuse [45, 59, 60], software evolution [16, 35], analysis [8, 53, 62, 121], and testing [26, 27]. Analyzing software architectures is a challenging task that has received significant attention from the research community during the past few years. Zhang et al. [121] have compared and classified over 16 approaches that analyze software architectures using model-checking techniques, which shows that this research topic is of growing interest. For example, Allen [9] applies model-checking techniques directly at the (software) architecture level, work that is later extended to also address the problem of specifying and analyzing dynamic architectures [7], whereas Burmester [40] proposes an approach for analyzing safety-critical, real-time systems. Compared to these contributions, which use a formal description and could support various kinds of formal verification, our framework is not limited to the use of exhaustive model checking and supports the use of model-based testing and statistical model checking. Thus our framework can be applied to a larger class of behaviors (e.g., infinite state behaviors).

In the automotive domain, two architectural languages have received special attention, EAST-ADL [33] and AADL (Architecture Analysis and Design Language) [54]. Several studies have been carried out towards the analysis of AADLs models enriched with its behavioral annex [25, 32]. Berthomieu et al. [25] use the Fiacre language for analyzing the functional behavior with the TINA tool. Our framework carries along the same goal: to provide a formal semantics to the architectural language, which in this case is AADL and not
EAST-ADL, for expressing timing behaviors. Yet, our work also targets other aspects of the model, which includes stochastic behavior and resource consumption. Thus, our framework is supporting more expressive models compared with the work mentioned above. To improve the usage of AADL in practice, Lasnier et al. [80] proposed an automatic code generation from AADL models and an analysis approach that is applicable to validate specific properties concerning real-time systems. Our work is not limited to code implementation and model-based testing, but rather we propose a wider range of analysis and verification techniques that target EAST-ADL models. In 2010, Bozzano et al. [34] have suggested the use of a model checker for AADL focusing on safety, dependability analysis and performance evaluation for AADL models in the aerospace domain. Compared to this work, our framework also relies on model-based testing to provide information on the correctness of the final implementation.

Several methods have been developed for the analysis of EAST-ADL models: Nallet et al. [65] describe the use of the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) together with EAST-ADL for timing analysis [65]. In the context of EAST-ADL UML2 profile, Feng et al. [55] proposed a translation of an existing behavioral representation of the system specified as activity diagrams in PROMELA and used the SPIN model checker for formal verification of EAST-ADL functional models. Qureshi et al. [102] 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. Sandberg et al. [106] provide an approach that performs iterative analysis of the safety architecture at analysis level to ensure that it still meets function specific safety goals after changes in the architecture. Compared to the above mentioned frameworks, our work provides exhaustive verification and statistical analysis to EAST-ADL, which together can check not only functional but also extra-functional properties, such as end-to-end deadlines and resource usage. Also, none of the works mentioned above target the implementation of these systems, or aim to provide a model-based testing framework that starts with the architectural model.

The idea of integrating analysis and testing has been discussed by Nielsen [97], in an attempt to increase the quality of embedded systems and to reduce development cost. He proposes a general, theoretical method for combining analysis and testing techniques, centered on a common verification strategy and iteratively exploiting the established results to strengthen the verification activities. The method proposed by the author is reflected in the framework
proposed in this thesis, where the same formal model is used for both the formal verification and model-based test generation. This notion, of integrating model-based analysis and testing has led to the extension of the UPPAAL tool chain with model based testing capabilities [75, 79], by also including a test-case generation method in the toolchain. Compared to our work, this is a general framework based on timed automata models and does not support test-case generation for UPPAAL PORT TA models. In addition, our framework targets the particularities of the EAST-ADL architectural language. Aichernig et al. [5] proposed a similar combined analysis and testing technique approach, where test cases (rather than models) are used in order to integrate different model-based testing and analysis tools. This work extended an already existing testing framework [4, 78], in contrast to our framework where the model-based testing method came as a continuation of the verification framework.

Testing software based on its architectural representation has received less attention than the analysis of such models. In 1996, the CHAM formalism was used to introduce the idea of software integration and unit testing based on the architectural model [27, 103]. Muccini and Bertolino have studied further this topic by performing different case studies that show the insight gain and some issues in architecture-based software testing [26, 95, 28]. Specifically, their research shows how to identify suitable tests for complex real-world systems. Other works have shown that architectural models can be used to automate generation of system level tests [2], to define architectural-based testing criteria [72, 103], and for regression testing [70]. The combination of analysis and testing of architectural languages has 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 [29], in which the author 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 descriptions, based on model checking techniques that can potentially be used in an industrial setting. However, we are aware that our method cannot be adopted by industry, as such: the formal models used by our framework need to be designed by specialists, meaning that engineers with knowledge of automata and logic need to be involved in the engineering process, which currently can be unfeasible in some industrial domains. Therefore, we have started working (and achieved first results [100]) on automated transformations between Simulink, a language frequently used in the development of industrial systems, and priced timed automata, a model amenable to statistical model checking. If successful, such an endeavor would
hopefully remove the mentioned limitation of our current framework.

In the area of architectural dependency analysis, Stafford et al. [110] have proposed a technique that identifies different types of direct dependencies for the Rapide [85] architectural models, focusing on the structural or the behavioral aspects, in order to compute the existing dependency chains within the model. These dependency chains can be analyzed in order to reveal errors early in the development process. Zhao [122] have proposed an architectural dependency analysis technique that uncovers high-level architectural dependencies of a software system modeled in ACME by providing the software architectural dependency graph that explicitly represents various types of dependencies in the architecture. Li [83] and Vieira et al. [115] also proposed a matrix-based approach for analyzing and managing dependencies in component-based systems. Other methodologies integrate different types of analysis. Gu et al. [67] have developed the AIRES tool, which provides dependency anomaly detection for ESML models [74], together with visual displays of dependency graphs, assignment of execution rates to component ports, timing and schedulability analysis, automated allocation of components to processors, etc. Garousi et al. [64] proposed a behavioral dependency analysis technique for UML representations of distributed systems. This technique assigns an index for any pair of system entities (objects, nodes, and networks) and defines a behavioral dependency analysis model to visualize dependencies in distributed systems. Nord et al. [98] use dependency analysis techniques and metrics to provide early information about the architectural trade-offs and efforts for creating a shared-computing resource architecture. In comparison to the above methodologies, we investigate the state-space reduction achieved in verification, based on existing dependencies, and we also validate the obtained model on an industrial prototype.
Chapter 7

Conclusions and Future Work

In this thesis, we have proposed a framework for the analysis and verification of automotive systems modeled in EAST-ADL. The framework provides behavior for EAST-ADL architectural models as timed automata networks in order to enable the formal analysis of such models based on different model-checking techniques. In our work, we employ: (i) UPPAAL, a well-known model checker for real-time systems, (ii) UPPAAL PORT, an extension of the UPPAAL model checker for component-based systems, and (iii) UPPAAL SMC, an extension of UPPAAL for statistical model checking. The UPPAAL 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. UPPAAL SMC is based on statistical model checking that, unlike symbolic model checking, can provide results only with a specified level of confidence, yet it is a cheap way to generate and confirm safety counter-examples where symbolic techniques may employ expensive over-approximations. Our verification methodology relies both on symbolic and statistical model-checking, since these techniques complement each other. Furthermore, the framework enables the formal verification of both functional and extra-functional system requirements, such as end-to-end deadlines and resource consumption.

In order to extend the verification framework with testing capabilities, we have investigated the possible needs and gaps of the current model-based testing methods used by industry and academia, and we have performed a liter-
nature review on the state of the art in the area of requirements-driven model-based testing. To achieve this goal, we have extended an existing model-based testing taxonomy with two new explicit dimensions: the test artifact (i.e., functional behavior, extra-functional behavior, or the architectural description), and mapping of tests (i.e., executable or abstract tests). One of our findings relates directly to the test artifact: while most model-based testing tools focus on generating test cases for functional properties, only a fraction focus on extra-functional properties (i.e., timing properties) and none of the considered model-based testing tools deal directly with architectural artifacts.

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

The work presented in this thesis provides concrete results on the feasibility of analyzing EAST-ADL architectural models by model checking techniques. The framework, which is supported by the ViTAL tool, 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. We have carried an extensive study on the feasibility of the proposed method by applying the ViTAL tool on the Brake-by-Wire industrial system prototype developed by AB Volvo. The results show that the framework is suitable for handling formal verification of industrial architectural models, for both functional and extra-functional requirements. In addition, we have shown that the toolchain, ViTAL plus Farkle, provides a complete model-based testing framework in which architectural models can be tested against functional and timing requirements.

In order to ensure exhaustive verification for complex automotive systems, we have investigated other analysis techniques that could reduce the state-space. To this end, we have proposed a method for pruning EAST-ADL architectural models extended with UPPAAL timed automata semantics. The method relies on computing the dependency matrix from the semantic representation of the EAST-ADL model as a network of timed automata and parsing it into
dependency chains. The dependency matrix is used to prune the architectural model, keeping only the relevant dependency chains with respect to a given requirement. The pruning method has been validated on the Brake-by-Wire system. To exhaustively verify a given end-to-end deadline on this system, the UPPAAL model-checker needs to explore 5060336 states, while on the pruned model only 66740 states are explored. This shows a considerable state-space reduction of about 98%.

The proposed framework has its limitations. First, applying the pruning algorithm cannot guarantee that exhaustive verification is possible. Depending on the requirement, many (if not all) components of the EAST-ADL model could be included in the dependency chain and the verification might not scale due to the state-space explosion. Second, the framework still relies on manual editing the behavior of the EAST-ADL components, thus it cannot be directly adopted by industry. The formal models used by our framework need to be designed by engineers with knowledge of timed automata, a modeling language that is not commonly used in industry.

As future work, we intend to further validate our approach on other industrial use cases. We also intend to investigate how other models used in the development of automotive systems, such as Simulink, as well as requirements specification tools [86] can be integrated in our framework to provide a verification framework that could be considered for industrial adoption.


Bibliography


[27] A. Bertolino and P. Inverardi. Architecture-based software testing. 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 62–64. ACM, 1996.


[103] 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.


