ARCHITECTURE-BASED VERIFICATION OF DEPENDABLE EMBEDDED SYSTEMS

Andreas Johnsen

2013
Abstract

Quality assurance of dependable embedded systems is becoming increasingly difficult, as developers are required to build more complex systems on tighter budgets. As systems become more complex, system architects must make increasingly complex architecture design decisions. The process of making the architecture design decisions of an intended system is the very first, and the most significant, step of ensuring that the developed system will meet its requirements, including requirements on its ability to tolerate faults. Since the decisions play a key role in the design of a dependable embedded system, they have a comprehensive effect on the development process and the largest impact on the developed system. Any faulty architecture design decision will, consequently, propagate throughout the development process, and is likely to lead to a system not meeting the requirements, an unacceptable level of dependability, and costly corrections.

Architecture design decisions are in turn critical with respect to quality and dependability of a system, and the cost of the development process. It is therefore crucial to prevent faulty architecture design decisions and, as early as practicable, detect and remove faulty decisions that have not been successfully prevented. The use of Architecture Description Languages (ADLs) helps developers to cope with the increasing complexity by formal and standardized means of communication and understanding. Furthermore, the availability of a formal description enables automated and formal analysis of the architecture design.

The contribution of this licentiate thesis is an architecture quality assurance framework for safety-critical, performance-critical, and mission-critical embedded systems specified in the Architecture Analysis and Design Language (AADL). The framework is developed through the adaption of formal methods, in particular traditional model-checking and model-based testing techniques, to AADL. This is done by defining both formal verification criteria for AADL, i
Abstract

Quality assurance of dependable embedded systems is becoming increasingly difficult, as developers are required to build more complex systems on tighter budgets. As systems become more complex, system architects must make increasingly complex architecture design decisions. The process of making the architecture design decisions of an intended system is the very first, and the most significant, step of ensuring that the developed system will meet its requirements, including requirements on its ability to tolerate faults. Since the decisions play a key role in the design of a dependable embedded system, they have a comprehensive effect on the development process and the largest impact on the developed system. Any faulty architecture design decision will, consequently, propagate throughout the development process, and is likely to lead to a system not meeting the requirements, an unacceptable level of dependability, and costly corrections.

Architecture design decisions are in turn critical with respect to quality and dependability of a system, and the cost of the development process. It is therefore crucial to prevent faulty architecture design decisions and, as early as practicable, detect and remove faulty decisions that have not been successfully prevented. The use of Architecture Description Languages (ADLs) helps developers to cope with the increasing complexity by formal and standardized means of communication and understanding. Furthermore, the availability of a formal description enables automated and formal analysis of the architecture design.

The contribution of this licentiate thesis is an architecture quality assurance framework for safety-critical, performance-critical, and mission-critical embedded systems specified in the Architecture Analysis and Design Language (AADL). The framework is developed through the adaption of formal methods, in particular traditional model-checking and model-based testing techniques, to AADL. This is done by defining both formal verification criteria for AADL,
and a formal semantics for AADL. Model-checking of AADL models provides evidence of the completeness, consistency, and correctness of the model, and allows for automated avoidance of faulty architecture design decisions, costly corrections and threats to quality and dependability. In addition, the framework can automatically generate test suites from AADL models to test a developed system with respect to the architecture design decisions. A successful test suite execution provides evidence that the architecture design has been implemented correctly. Methods for selective regression verification are included in the framework to cost-efficiently re-verify a modified architecture design, after e.g., a correction of a faulty design decision.
Swedish Summary

Ett inbyggt system är ett datorsystem som har en specifik funktion inom ett större elektroniskt, och möjlichen mekaniskt, system. I motsats finns exempelvis persondatorer, som är designade att kunna användas till ett stort antal olika ändamål. Ett tillförlitligt inbyggt system är ett inbyggt system vars funktion dessutom är kritisk för både systemet i helhet och för omgivningen systemet agerar i. Exempel på tillförlitliga inbyggda system är elektroniska styrsystem i flygplan, datorsystem för flygledning, datoriserade styrsystem för kärnkraftverk, och bilens farthållare. Eftersom en felaktig funktion i dessa system kan vara kostsamma och leda till fara för människor och omgivning, är det avgörande att säkerhetsställa att ett inbyggt system uppnår alla kvalitetskrav ställda på dess funktion. Kvalitetssäkring av tillförlitliga inbyggda system är en ständigt växande utmaning eftersom utvecklare av sådana system är tvungna att bygga allt mer komplexa system inom allt mer begränsade budgetar vad gäller både tid och pengar.

Då systemens komplexitet ökar måste systemarkitekternas beslut om systemens arkitekturdesign. Dessa beslut är de första man tar i en utvecklingsprocess efter att problemet man vill lösa noggrant har analyserats. En arkitekturdesign är en övergripande teoretisk lösning på det problem man vill att det tilltänkta systemet skall lösa. I arkitekturdesignen ingår ider om vilka de väsentliga komponenterna (som till exempel sensorer, processorer, drivdorn, databussar, operativsystem, kommunikationsprotokoll och applikationer) i det tilltänkta systemet är, hur de skall vara strukturerade, och hur de skall interagera med varandra och med omgivningen. Då dessa beslut påverkar de övergripande egenskaperna av systemet har de stor effekt på utvecklingsprocessen samt den största påverkan på det, senare, utvecklade systemet. På grund av detta kommer ett felaktigt beslut vad gäller arkitekturdesignen breda ut sig genom hela utvecklingsprocessen och sannolikt resultera i ett system som inte uppnår de krav systemet måste uppfylla, får en
oacceptabel tillförlitlighetsnivå, och kommer kräva kostsamma korrigeringar. Beslut om en arkitekturdesign är därmed kritiska med hänsyn till kvaliteten och tillförlitligheten hos ett inbyggt system, samt kostnaden för utvecklingsprocessen. Således är det kritiskt att i största möjliga mån förhindra att felaktiga beslut tas gällande arkitekturdesignen och, så tidigt som möjligt, upptäcka och avlägsna felaktiga beslut som inte har lyckats förhindras.


Bidraget av denna licentiatavhandling är en datoriserad analyseteknik och en testningsteknik för tillförlitliga inbyggda system modelleras med hjälp av AADL. Teknikerna kan automatiskt både analysera AADL-modeller och testa om ett utvecklat system överensstämmer med dess AADL-modell. En automa-
tisk analys av en AADL-modell visar om några felaktiga designbeslut har tagits. Detta möjliggör automatisk undvikande av kostsamma korrigeringsar och hot mot kvalitet och tillförlitlighet. En automatisk testning av ett utvecklat system visar om det är korrekt byggt enligt dess AADL modell och kan tas i bruk. Eftersom teknikerna sannolikt kommer att användas upprepade gånger under en utvecklingsprocess, på grund av korrigeringsar och modifieringar av arkitekturdesignen, har de utökats med effektiveringsmetoder. Dessa metoder gör att en analys eller testning kan göras på så kort tid som möjligt.
Acknowledgments

First of all, I would like to thank my supervisors Prof. Kristina Lundqvist and Prof. Paul Pettersson for their support, encouragement and guidance throughout my research – this thesis would not be possible without you! It has been nothing but a great pleasure to work with you, and I am looking forward to the work we have ahead of us.

There are so many other people to thank. I would like to jointly thank all the people at the IDT department, Mälardalen University. I have truly not seen a more encouraging, inspiring and open-minded environment to work in.

Many special thanks to: Prof. Lars Asplund for encouraging me to become a Ph.D. student; my office-mates over the years, Adnan Causevic, Hüseyin Aysan, Etienne Borde, Jiale Zhou and Abhilash Thekkilakattil, for all the help and great office hours; Mehrdad Saadatmand, Federico Ciccozzi, Antonio Cicchetti and Thomas Leveque for all the fun we have had both at work, as well as outside of work; Kristina Forsberg and Göran Bertheau for their interest in my work, and the knowledge they have given me about avionic systems; and Stefan Björnander for the valuable knowledge and suggestions he has given me about AADL. I thank Mattias Nyberg for the collaboration with Scania, which has led to valuable research results, master’s theses and teaching material.

I would also like to thank, Hans Hansson, Thomas Nolte, Cristina Secelianu, Sasikumar Punnekkat, Ivica Crnkovic, Mikael Sjödin, Iain Bate, Radu Dobrin, Daniel Sundmark, Frank Lüders, Barbara Gallina, Kristian Wiklund, Patrick Graydon, Yue Lu, Andreas Gustavsson, Jan Carlson, Jan Gustafsson, Björn Lisper, Henrik Thane, Omar Jaradat, Mikael Åsberg, Aida Causevic, Leo Hatvani, Juraj Feljan, Kivanc Doganay, Aneta Vulgarakis, Moris Behnam, Saad Mubeen, Dag Nyström, Sèverine Sentilles, et al., for all the help, inspiration and interesting discussions. I thank the administrative staff, Malin Rosqvist, Carola Ryttersson, Gunnar Widforss, Susanne Fromnå, et al., for making many things easier.
Finally, I would like to express my deepest gratitudes to my beloved family, my friends outside of work, and my dearest, Dr. Nina Elisabeth Holt.

Andreas Johnsen
Västerås, June, 2013
List of Publications

Papers Included in the Licentiate Thesis

1. **Paper A**
   Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL.

2. **Paper B**
   An Architecture-Based Verification Technique for AADL Specifications.

3. **Paper C**
   Automated Verification of AADL-Specifications Using UPPAAL.

4. **Paper D**
   Architecture-Based Regression Verification of AADL Specifications.

1The included articles have been reformatted to comply with the licentiate layout.
List of Publications

Papers Included in the Licentiate Thesis


---

1The included articles have been reformatted to comply with the licentiate layout.
Related Publications not Included in the Licentiate Thesis


- **Industrial Experiences of Building a Safety Case in Compliance with ISO 26262.** Raghad Dardar, Barbara Gallina, Andreas Johnsen, Kristina Lundqvist and Mattias Nyberg. Proceedings of the ISSRE 2012 2nd Workshop on Software Certification (WoSoCER’12), Dallas, TX, USA, November, 2012.
## Contents

I  Thesis  

1 Introduction  
1.1 Motivation  
1.2 Contributions  
1.3 Thesis Outline  

2 Background  
2.1 Model-driven Engineering  
2.2 Model-checking and the UPPAAL Model-checker  
2.3 Model-based Testing  
2.4 Regression Verification  
2.5 The Architecture Analysis and Design Language  
2.5.1 Component Type  
2.5.2 Component Implementation  

3 Research Overview  
3.1 Research Challenges and Goals  
3.2 Related Work  
3.2.1 Evaluating ADLs  
3.2.2 Architecture-based Verification  
3.2.3 Formal Semantics for Formal Verification and Model-Based Testing  
3.3 Research Methodology  
3.4 Summary of Results  
3.4.1 Contribution 1 – A Comparison Framework  
3.4.2 Contribution 2 – A comparison of AADL and EAST-ADL
3.4.3 Contribution 3 – An Architecture-Based Verification Technique ............................................. 33
3.4.4 Contribution 4 – A Formal Semantics of a Subset of AADL .................................................. 34
3.4.5 Contribution 5 – Validation of the Verification Technique ....................................................... 35
3.4.6 Contribution 6 – Regression Verification ................................................................................. 35

4 Conclusion ......................................................................................................................................... 37
4.1 Future Work .................................................................................................................................. 38

Bibliography ........................................................................................................................................ 41

II Included Papers ............................................................................................................................. 47

5 Paper A:
Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL ........................................ 49
5.1 Introduction .................................................................................................................................... 51
5.2 The Comparison Framework ....................................................................................................... 52
  5.2.1 Building Block: Component ..................................................................................................... 52
  5.2.2 Building Block: Connector ....................................................................................................... 54
  5.2.3 Building Block: Configuration .................................................................................................. 54
  5.2.4 Vital Quality Attributes ............................................................................................................ 55
5.3 ADLs Under Comparison ............................................................................................................. 56
  5.3.1 Overview of AADL ..................................................................................................................... 56
  5.3.2 Overview of EAST-ADL ............................................................................................................ 56
5.4 AADL vs. EAST-ADL .................................................................................................................. 57
  5.4.1 Modeling of Components ........................................................................................................ 57
  5.4.2 Modeling of Connectors .......................................................................................................... 61
  5.4.3 Modeling of Configurations .................................................................................................... 62
  5.4.4 Dependability .......................................................................................................................... 65
  5.4.5 Timing ...................................................................................................................................... 66
5.5 Conclusion ...................................................................................................................................... 66
Bibliography .......................................................................................................................................... 69

6 Paper B:
An Architecture-Based Verification Technique for AADL Specifications ............................................. 71
6.1 Introduction ..................................................................................................................................... 73
6.2 Preliminaries ....................................... 74
6.3 The Architecture-based Verification Technique .......... 75
6.4 AADL Verification Criteria .............................. 76
   6.4.1 Verification objectives ............................ 77
   6.4.2 Verification Criteria ............................... 79
6.5 Conclusion ............................................ 81
Bibliography ............................................. 83

7 Paper C:  
Automated Verification of AADL-Specifications Using UPPAAL 85
7.1 Introduction ........................................... 87
7.2 AADL and The Verification Technique .................... 88
7.3 Transformation to Timed Automata ......................... 92
   7.3.1 Transformation Rules .............................. 94
7.4 Case study ............................................. 102
7.5 Related Work .......................................... 103
7.6 Conclusion ............................................. 104
Bibliography ............................................. 105

8 Paper D:  
Architecture-Based Regression Verification of AADL Specifications 107
8.1 Introduction ........................................... 109
8.2 Background ........................................... 111
   8.2.1 The Architecture Analysis and Design Language ... 112
8.3 Selective Regression Verification Through AADL Model Slicing 112
   8.3.1 Overview of the Slicing Algorithm .................. 113
   8.3.2 Generation of CFGs ................................ 118
   8.3.3 Generation of the PDG Through a CDG and a FDG .... 122
   8.3.4 Generation of the SDG .............................. 124
   8.3.5 Calculating the Slice for Selective Regression Verification ............. 128
8.4 Conclusion ............................................. 128
Bibliography ............................................. 129
I

Thesis
Chapter 1

Introduction

The increasing complexity of dependable embedded systems is challenging the already restricted budget constraints of the development process. Dependable embedded systems are required to deliver services that can be justifiably trusted [1]. The use of methods for quality assurance in the development process is essential to meet this requirement, and typically stands for a majority of the development cost. With an increasing complexity, the ability of assuring quality within already restricted budgets becomes even harder.

Architecture-based development approaches promise to provide means for reducing the cost of the development process of complex dependable systems while increasing the quality and the dependability of the developed system.

Architecture-based development is a generic term including the following procedures: eliciting the architectural requirements, designing the architecture, documenting/modeling the architecture, analyzing the architecture, realizing/implementing the architecture, and maintaining the architecture. The paradox of reducing cost while increasing quality is partially explained by the fact that system architecture models allow developers to reason about the system from a perspective closer to the problem domain and at an appropriate level of abstraction. These abilities result in a more predictable development process, where incorrect design decisions may be avoided that otherwise would lead to costly corrections, or worse, to operational failures [2, 3, 4].

Numerous so called architecture description languages (ADLs) have been developed due to the increasing need for architecture-based development approaches. ADLs are tailored to represent architecture design decisions, and
Chapter 1

Introduction

The increasing complexity of dependable embedded systems is challenging the already restricted budget constraints of the development process.

Dependable embedded systems are required to deliver services that can be justifiably trusted [1]. The use of methods for quality assurance in the development process is essential to meet this requirement, and typically stands for a majority of the development cost. With an increasing complexity, the ability of assuring quality within already restricted budgets becomes even harder. Architecture-based development approaches promise to provide means for reducing the cost of the development process of complex dependable systems while increasing the quality and the dependability of the developed system. Architecture-based development is a generic term including the following procedures: eliciting the architectural requirements, designing the architecture, documenting/modeling the architecture, analyzing the architecture, realizing/implementing the architecture, and maintaining the architecture. The paradox of reducing cost while increasing quality is partially explained by the fact that system architecture models allow developers to reason about the system from a perspective closer to the problem domain and at an appropriate level of abstraction. These abilities result in a more predictable development process, where incorrect design decisions may be avoided that otherwise would lead to costly corrections, or worse, to operational failures [2, 3, 4].

Numerous so called architecture description languages (ADLs) have been developed due to the increasing need for architecture-based development approaches. ADLs are tailored to represent architecture design decisions, and
provide the possibility to develop tools for automated and formal verification. However, there are still open questions on:

- the right choice of ADL for dependable embedded systems,
- how formal verification based on architectures represented by ADLs can allow for a higher trust in both the development process as well as the developed system, and
- how to perform formal verification based on architectures represented by ADLs automatically.

In this thesis, we seek answers to these questions and contribute with a holistic architecture quality assurance framework for the development of dependable embedded systems represented by the ADL Architecture Analysis and Design Language (AADL) [5] – an ADL we in Paper A (Chapter 5) show is suited for the development of dependable embedded systems. The framework includes techniques for model-checking, moebel-based testing and regression verification of AADL models. The framework, thereby, covers the entire development process, from the very first architecture design decisions modeled in AADL to the developed system and subsequent modifications of the architecture.

1.1 Motivation

Dependability is a generic concept that subsumes the following quality attributes (also known as non-functional or extra-functional properties): availability, reliability, safety, integrity, and maintainability [1]. A dependable embedded system is a system with a dedicated function within a larger electrical and/or mechanical system, where software interacts with sensors, actuators, devices, other systems, and people, and where a combination of availability, reliability, safety, integrity and maintainability is required from the system. Examples of such systems are fly-by-wire control systems, air traffic management systems, and automotive Electrical/Electronic (E/E) systems. These systems are typically

- mission-critical: a correctly provided service is an absolute necessity for an organization in order to execute its mission.
- safety-critical: a deviation from correct service may endanger the environment and people.
1.1 Motivation

- and performance-critical: services that are provided outside the real-time constraints are considered as system failures.

These characteristics demand stringent requirements of high dependability and quality. We define quality as the conformance to documented and logically implied functional and non-functional requirements that are actionable, measurable, traceable, testable, complete and consistent.

In essence, the threat to quality and dependability is faults. A fault can take various forms, ranging from development faults, such as software design faults, to operational faults, such as random hardware faults. The use of methods for fault avoidance in the development process is essential to achieve dependability and quality [6]. Fault avoidance is a term used to represent methods focused on developing a fault-free system. It includes both proactive methods, known as fault prevention, which prevent faults from being introduced, and reactive methods, known as fault removal, which remove faults that have been introduced. Although the development of a fault-free system seldom is practical and has to be supported by techniques, known as fault tolerance, retaining dependability even in the presence of faults, fault avoidance is necessary to assure dependability and quality. This is necessary since even fault tolerance techniques themselves must be developed with fault avoidance to be acceptably dependable. The architecture model is one of the most critical artifacts in the development process of dependable embedded system with respect to both fault avoidance and fault tolerance [4]. The term architecture model is used interchangeably with architecture specification throughout the thesis.

A development lifecycle has a set of stages that generally include requirements analysis, design, implementation and maintenance. These stages are typically carried out in the corresponding sequence, often with a number of iterations and an amount of overlaps between them. Requirements analysis is carried out to understand the needs of the customer(s), user(s), and other stakeholders such as managers, certification authorities, investors, developers, suppliers, etc. The creation of the architecture model, which is a representation of the design decisions made on the architecture of the intended system, is the very first step of ensuring that the system will meet the requirements of the stakeholders [4]. The process of making the architecture design decisions involves the allocation of functional properties to certain architectural structures and patterns (known to exhibit certain quality attributes) such that the required quality attributes, such as performance, reliability, safety, modifiability, testability, security, etc., are achieved. The process includes allocation of functions to structures of hardware, software, information and/or time re-
dudnance to construct fault tolerance mechanisms. The architecture model is subsequently used as a blueprint among stakeholders and serves as a basis for the entire development process, where the design decisions will be refined with more details until it is ready to be implemented. Hence, the end product will heavily depend on the architecture model, which it should conform to.

Due to the fact that architecture design decisions have the most far-reaching effects on the development process, they are the most critical design decisions to analyze [4]. A design fault introduced by incorrect, inconsistent, or incomplete architecture design decisions will propagate throughout the refinement of the design, and is therefore likely to have a deteriorating impact through the complete development process. In addition, since a fulfillment of the developed system with respect to the requirements of stakeholders is dependent on the architecture design decisions, it is critical to test that they have been implemented correctly. A fault introduced in the process of refining the architecture design decisions, or in the implementation of them, results in an end product not conforming to the architecture design decisions, and is likely to produce a system with unanticipated quality attributes, including dependability.

A development lifecycle typically includes a set of Verification and Validation (V&V) activities to ensure that the system is built right (verification), and that the right system is built (validation). Validation often involves subjective judgements from stakeholders, and typically includes requirements analysis and system testing. Verification is typically performed by three different approaches: 1) reason about and analyse the system through formal methods and static analysis (model-checking/model verification, theorem proving, abstract interpretation, data-flow analysis, simulation, etc.); 2) inspect and review artifacts; and 3) experiment with the system through dynamic analysis (all types of testing, including model-based testing). Verification typically stands for a majority of the total development cost of software systems due to the requirements of high quality [7]. Moreover, empirical studies show that the cost of finding and correcting faults dramatically increases the later they are found in the lifecycle [8], and the majority of faults are introduced by incomplete and inconsistent requirement and design specifications and models [9].

The increasing complexity of dependable embedded systems in combination with the criticality of architecture design decisions and the cost of late fault corrections have evolved a need for approaches supporting developers with the understandability, the communication, and the analysis of architecture design decisions [10]. In [11], Elm et al. present a survey of projects, executed by defense contractors, that quantifies the relationship between System Engineering (SE) best practices and the performance of the projects in terms of
cost and schedule. Results show that there is a strong positive relationship between architecture-based development capabilities and the performance of the projects. For example, only 11% of the projects with lower architecture-based development capabilities exhibited a good performance. Furthermore, Boehm et al. [12] quantitatively present the return on investment of system engineering based on an analysis of 161 projects. Results partly show that 20% of the defects account for 80% of the rework costs, and that these 20% of defects primarily came from an inadequate architecture definition and risk resolution.

Model-Driven Engineering (MDE) [13], or Model-Driven Development (MDD), is a promising approach abstracting the complexity of systems, and enables powerful analysis in the early phases of the development process such that faults can be prevented and removed as early as possible [14, 15]. The approach has partly lead to the development of Architecture Description Languages (ADLs) to support understandability, communication, and analysis of architecture design decisions by standardized and formal means. Communication, understandability and analysis of architecture design decisions are closely related but variably supported by different ADLs. AADL has the ability to describe dependable real-time embedded systems and supports all these aspects, partly by a “precise” semantics. The semantics of AADL is defined in a natural language and therefore inherently prone to misinterpretations. The semantics is however precise from an informal point of view due to the extensive amount and diversity of details. A precise semantics does also facilitate a formalization of the semantics, which is suitable for automated formal verification and model-based testing. In our case study [16], involving six international companies developing safety-critical embedded systems in the vehicular and avionics domains, the results showed that formal verification is ranked as one of the most important type of quality assurance techniques. The usage of formal methods when performing verification is becoming increasingly important in the certification of dependable systems due to the increasing demand for evidence based on well-founded mathematical principles. The importance of formal verification in the development of dependable systems is not a new discovery. For example, Bowen and Stavridou [17] confirmed the industrial need for formal methods for verifying safety-critical systems in the early 90s. Moreover, the disadvantages of manual verification, such as the human risk and the time consumption, are crucial to overcome to be able to develop more dependable systems at a lower cost. Formal methods, such as model-checking and theorem proving, and extensions thereof, such as model-based testing (an integration of formal methods and dynamic verification), allow for automated verification through powerful and mature computer tools.
Automated formal verification of architecture design decisions, and model-based testing from architecture models, in turn, have the potential to significantly reduce the cost of the development process while increasing the quality and dependability of the end product. In addition, numerous research efforts have been devoted to the development of more efficient regression testing techniques, as studies have shown that regression testing consumes a considerable part of the verification cost of software systems [18]. Several research efforts conducted within the area of regression testing have developed techniques for determining the minimal subset of a regression test suite necessary to retest a modified software system [19]. Although the efficiency of regression testing is highly important, architecture models are, in addition to a source vulnerable to fault introduction, also subjected to a large number of modifications. Hence, they are also subjected to regression verification activities in form of reviews, inspection, static analysis, model-checking, model validation, simulation, etc. The techniques used to perform regression verification of architecture models do seldom consider previous verification executions and the effect prior modifications have on the artifact(s) under analysis. As the information is disregarded, it is impossible to track the coverage of each artifact for each of the previous verification sequence executions, and to track the modified and affected elements of the artifact(s) necessary to re-verify. These are two important problems in selective (efficient) regression verification [19]. A complete rerun of the verification activities is consequently necessary to ensure that no new faults follow from a modification. Such reruns can be extensively time-consuming and costly depending on the extent of the involved artifacts. A stronger emphasis on the analysis of the evolution of architecture models through modifications, therefore, also has the potential to significantly reduce the verification cost of dependable embedded systems. The need for such techniques, though for software in general, was discovered by Harrold [7] in the beginning of the 21st century. ADLs partly provides standardized and formal means upon which effective and efficient regression verification techniques can be developed.

1.2 Contributions

In this thesis, we propose a holistic architecture quality assurance framework for the development of dependable embedded systems specified in the Architecture Analysis and Design Language (AADL) [5] developed by the Society of Automotive Engineers (SAE). The language supports a hierarchical
component-connector paradigm and is able to represent both software and hardware architectures of safety-critical, mission-critical, and performance-critical embedded systems. First, in Paper A (Chapter 5), we show why AADL has the ability to meet most qualities of an ideal ADL for dependable embedded systems and compare it with an alternative ADL through a developed comparison framework. Second, in Paper B (Chapter 6), we present a quality assurance approach covering the entire development process, from the very first architecture design decisions specified in AADL to the architecture of the developed system. The approach is developed through an adaptation of the theoretical knowledge within the areas of formal verification through formal methods, and automated testing from specifications (model-based testing), to an architectural perspective. Through the adaption, we are able to address the automated analysis of architecture design decisions specified in AADL and the testing of a developed system with respect to the architecture design decisions – these activities play a major role in the development of dependable embedded systems as described in Section 1.1.

Due to the features of an architecture, the primary focus of analysis at the architecture-level is the integration of components. We use the verification objective of traditional integration testing to analyze AADL models. The objective is to ensure consistency and completeness of component interfaces and to ensure that data and control are passed correctly among components. By formally defining the possible control and data flows among components in AADL, diagrams representing the flows of a system can be extracted from any AADL model. Control flows refer to the order in which software elements are executed, whereas data flows refer to the orders in which data elements are defined/assigned values and how these assignments affect subsequent readings/uses of the data elements. Based on such diagrams, all the possible paths of execution that are necessary to be analyzed can be extracted. In addition, these paths can be used to generate test cases for the implementation to test its conformity with the AADL model. Third, we are able to carry out these activities automatically by defining, in Paper C (Chapter 7), a formal and implemented semantics of a subset of AADL in timed automata. A definition in timed automata allows for the usage of the UPPAAL model checker [20], where the analysis activities can be automated through model-checking and model-based testing. The paths of execution generated from the control and data flow graphs can then be mapped to the corresponding timed automata paths, where temporal logic is used to verify that the specification correctly passes data and control. The exercise of a path generates a trace containing preconditions, postconditions, and the timing constraints of the system. These
traces are used to generate test cases to test the architecture of the implemented system. Finally, we integrate the architecture-based verification technique with a method for efficient regression verification defined in Paper D (Chapter 8). The methods is based on comparisons and slicing [21] of control and data dependency graphs of an AADL model and its modified version. Dependencies that are not affected by the modifications can then be identified and disregarded in the regression verification process to avoid unnecessary costs.

1.3 Thesis Outline

The thesis is organized into two parts: Part I includes four chapters. Chapter 1 has provided an introduction to the thesis where an overview of the research problem, the motivation and the thesis contribution were presented. In Chapter 2, we present the background information on model-driven engineering, model-checking, model-based testing, regression verification (including regression testing), and AADL. In Chapter 3, a research overview is presented, including detailed definitions of research challenges and goals, related work, research methodology and summary of results. In Chapter 4, we summarize the work with concluding remarks and suggestions for future work. Part II contains the research papers included in the thesis, which are organized as:

**Chapter 5 (Paper A):** Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL

**Chapter 6 (Paper B):** An Architecture-Based Verification Technique for AADL Specifications

**Chapter 7 (Paper C):** Automated Verification of AADL-Specifications Using UPPAAL

**Chapter 8 (Paper D):** Architecture-Based Regression Verification of AADL Specifications
Chapter 2

Background

2.1 Model-driven Engineering

The basic principle of the MDE methodology is that a system is represented by one or several models, each of which conforms to a meta-model [22]. Metamodels, which define the rules a proper model must conform to, impose one or multiple viewpoints of a system. The rules are most often solely related to the form (syntax) of the language, however, in this thesis, the rules are also related to the meaning (semantics) of the language. A viewpoint abstracts particular concerns of a system to achieve the notion of separation of concerns [23]. The application of a viewpoint to a system generates one or several models which represent(s) the system from the perspective of the corresponding set of concerns. AADL models represent real-time embedded systems from both software- and hardware-architectural viewpoints at different levels of abstraction. Models may be both prescriptive, i.e., representing a system that is intended to be realized, or descriptive, i.e., representing a system that already is realized. In each case the model can be effectively and efficiently used for cognitive purposes, such as understanding, communication, analysis, synthesis, and prediction, with respect to using the definitely more complex, expensive, and in some cases unsafe real system [24]. Based on the model, vital (theoretical) conclusions can be drawn from the inference rules of the formalism.

In essence, the benefits of practicing the MDE methodology rely on its basic principle, that is, the prescriptive/descriptive model conforms to its meta-model and the prescriptive/descriptive model represents the intended/existing system. As soon as a model drifts out of conformance of its meta-model, or
does not represent the intended or existing system, many of the conclusions drawn during the development process become invalid. Much research has been devoted to provide means of (partly) ensuring that the basic principles are followed, such as syntax-checkers, model-checkers, semantic-preserving model-transformations, model compilers, model-based testing tools, traceability tools, etc.

Two goals of the verification technique we propose in this thesis are partly to support the two basic principles of MDE with respect to AADL models. More specifically, the first goal, which is to support the conformance of a model to its meta-model, is to verify correctness, completeness, and consistency of AADL models based on the semantic rules of the language. The technique assumes the AADL model to be syntactically correct. There exist numerous syntax-checkers for AADL, such as the front-end processing toolset provided by the OSATE (Open Source AADL Tool Environment) Eclipse IDE [25], hence, syntactical correctness is not considered as a challenge in our work. The second goal, which is to support the representation of a system by a model, is to verify the end-system against its AADL model (also known as model validation).

2.2 Model-checking and the UPPAAL Model-checker

Model-checking refers to the verification of a model of a finite state concurrent system in a precise mathematical language against a requirement specification of logic formulae [26]. The modeling language is usually based on the automata theory. In this thesis, we are using the model-checking tool UPPAAL [27] to model-check AADL models. We have chosen UPPAAL due to its ability to verify timing constraints, which often are critical constraints of dependable embedded systems. In UPPAAL systems are modeled by a network of timed finite state automata and checked by a subset of Timed Computational Tree Logic (TCTL) [20]. A timed finite state automaton consists of locations (nodes), edges (arcs) connecting locations, and labels (alphabet). UPPAAL extends the automata theory with clock variables to model time, where all clocks progress synchronously through real numbers, and with discrete variables that can be read, assigned, or used for arithmetic operations. Edges are labeled with events in terms of synchronizations, timing constraints in terms of constraints on clock variables, and data variables in terms of transition guards and variable assignments. A system is modeled as a network of timed automata, and
transitions in different automata can be synchronized using so called channels. Locations can be decorated with invariants and stated as urgent or committed. Invariants are atomic properties (e.g. \( x < 10 \)) that invariantly hold within the locations they are decorating. In an urgent location, time is not allowed to progress, whereas in a committed location, time is not allowed to progress and the next transition must involve one of its outgoing edges. A state of a system is defined by “the locations of all automata, the clock constraints, and the values of the discrete variables” [20]. A path is a sequence of states such that there exist edges that relates each state to its successor state.

As an example of an UPPAAL model, we will look at the so called Vikings problem that comes with the UPPAAL install files [27]. Four Vikings with one torch are supposed to traverse a damaged bridge during night in order to get away from the unsafe side to the safe side of the bridge. Since it is dark and the bridge is damaged, a maximum number of two Vikings can traverse the bridge simultaneously by using the torch. The Vikings traverse the bridge in different time units, where Viking 1 needs 5 time units, Viking 2 needs 10 time units, Viking 3 needs 20 time units and Viking 4 needs 25 time units. Since there is only one torch, two Vikings traverse the bridge together, where the time needed is equal to the slowest Viking. Note that there is only one torch, hence, the torch must be carried back by a Viking such that two more Vikings can traverse the bridge to the safe side, and so forth. The underlying question to the problem is if all Vikings can be at the safe side within 60 time units. An UPPAAL model of this system is partly shown in Figure 2.1.

- Automaton \( A \) represents one Viking (each Viking is modeled by an automaton similar to \( A \) where it differs only by the delay variable) and automaton ”B” represents the torch.

- The automata synchronize by two channels "take" and "release", and represent a Viking grabbing or releasing the torch. A channel name with an exclamation mark represents the sender where a question mark represents the receiver, i.e. take! can synchronize with take?.

- \( L \) is a global integer variable (reached by all automata) that can be assigned the values 0 or 1, and represents which side the torch is at. 0 represents the unsafe side whereas 1 represents the safe side.

- \( y \) is a local clock (reached by one automaton/Viking) that is used with the local integer constant delay to represent the time needed to traverse the bridge. delay is assigned the value 5, 10, 20 or 25 depending on which Viking the automaton represents.
• *unsafe*, *safe*, *free*, *two* and *one* are identifiers of locations where the *unsafe* and *safe* locations represent which side of the bridge the Viking is at. Locations *free*, *two* and *one* represent the torch being held by zero, one or two Vikings.

![Figure 2.1: The Viking automaton](image1)

A model is verified by TCTL queries in the form of path formulae and state formulae. The UPPAAL model-checker searches the state-space of the model to check if it satisfies the formulae. State formulae are expressions that describe properties of individual states while path formulae are expressions that describe properties over paths of states. A state formula is a predicate such as $x = 4$ or $x \leq 10$ where these formulae are valid in a state whenever $x$ equals four or $x$ is less or equal to ten. State formulae can be evaluated to valid or invalid for a state without analyzing the behavior of the model to or from the particular state. Path formulae are further classified into reachability, safety, and liveness property formulae: a reachability property formula checks whether a predicate

![Figure 2.2: The torch automaton](image2)
can be satisfied by a reachable state along some path; a safety property formula checks whether a predicate invariantly is satisfied in each state of a path, or all paths; and a liveness property formula checks whether a predicate eventually is satisfied by a reachable state in all paths.

Reachability properties are verified using temporal operators $E$ (pronounced for some path or exists one path) and $<>$ (pronounced eventually). For example, in order to verify if a state formula $p$ is reachable, we simple check it by the formula $E <> p$ (pronounced for some path eventually $p$ holds). The Vikings problem described above can now be checked by using the reachability formula $E <> (\text{Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe and time} <= 60)$ where $\text{time}$ is a global clock never reset in the model. Safety properties are verified using temporal operators $A$ (pronounced for all paths), $E$, and $[]$ (pronounced always or globally). Formula $A [] p$ (pronounced for all paths globally $p$ holds) is used if the property should hold in all states for all paths whereas formula $E [] p$ (pronounced for some path globally $p$ holds) is used if the property should hold for all states in at least one path. Liveness properties are verified by using temporal operators $A$, $<$>, and $\rightarrow$ (pronounced leads to). Formula $A <> p$ (pronounced for all paths eventually $p$ holds) checks whether $p$ eventually holds in all paths whereas $p \rightarrow q$ (pronounced whenever $p$ holds eventually $q$ holds) checks whether $q$ eventually holds whenever $p$ holds. The Uppaal query language is extended with a special deadlock state formula that can be used with reachability and safety formulae.

### 2.3 Model-based Testing

V&V has traditionally been carried out through manual or semi-automated testing. Manual testing is usually conducted by reading requirements and system specifications in order to directly experiment with the system under test (SUT). Semi-automated testing is usually conducted by reading requirements and system specifications in order to manually design test cases that can be encoded into test scripts. Test scripts can then be automatically executed against the SUT. Even though test scripts can be automatically executed against the SUT for initial or regression testing, the disadvantages of manual testing, i.e., the human risk (e.g., missed tests, incorrect tests, redundant tests, unknown requirement coverage, etc.) and the time consumption, are crucial to overcome.

A growing research effort has been carried out in the field of automated testing through model-based testing (MBT) to solve these problems, enclosing different levels of testing, such as unit, integration and system testing, as well as the
different objectives of testing, such as acceptance, performance, functional, reliability, and regression testing [28]. MBT is a collection of approaches where computers automatically perform black-box software testing from computer readable models. Software can be tested by models from two different perspectives, either by so called test models representing the user and the environment interacting with the system, or by so called system models representing the expected behavior of the SUT. In either case, the model comprises an input domain, an expected output range and mappings between the two. Each mapping can therefore serve as a test case.

Models for MBT are commonly represented by notations based on the automata theory [29], where test cases can be automatically generated by computer tools according to some coverage criteria, such as state coverage (covering all states), transition coverage (covering all state transitions), condition/predicate coverage (covering all truth and false values of all conditional constructs) and path coverage (covering all possible paths). Test cases are generated by searching the state-space such that traces adequately cover the coverage criteria. Each trace includes an input, or a sequence of inputs, and the expected output, or sequence of outputs, which can be formed into test cases. An adaptor software component between the model and the SUT is typically necessary to execute the generated test cases due to the different abstraction levels of the generated test cases and the SUT [29]. Thus, input of test case must be concretised such that it conforms to the interface of the SUT, and output produced by the SUT must be abstracted to verdict if it matches the expected output.

### 2.4 Regression Verification

A development process is most likely to include modifications of artifacts that previously have undergone verification, including the developed system, for the purpose of improvement and correction. Verified artifacts that subsequently are modified must necessarily be re-verified to ensure that no faults have been introduced in response to the modification. We use the term regression verification as a collective term for this type of verification, including the thoroughly studied area of regression testing [30]. Regression testing is commonly approached by testing the modified part of a software system and reusing previously executed test cases to test if the modification has introduced faults in previously functioning software [19]. Since a modification may not affect every functional and non-functional property of the system, it may not be neces-
2.5 The Architecture Analysis and Design Language

AADL was initially released and published as a Society of Automotive Engineers (SAE) Standard AS5506 [33] in 2004, and a second version (AADLv2) [5] was published in 2009. It is a textual and graphical language used to model, specify, and analyze software- and hardware-architectures of real-time embedded systems. AADL is based on a component-connector paradigm that hierarchically describes components, component interfaces, and the interactions (connections) among components. Hence, the language captures functional properties of the system, such as input and output through component interfaces, as well as structural properties through configurations of components, subcomponents, and connectors. Furthermore, means to describe quality attributes, characteristics, and constraints, such as timing and reliability, of application software and execution platform components are also
provided through explicit property associations. Changes to the runtime architecture can be described by modes and transitions of modes, and behavior of components can be described by state transitions systems defined in the Behavioral Annex (BA) [34]. AADL defines component abstractions which are further divided into three groups:

**Application software components**

- **Process component**: represents a protected address space containing at least one thread.
- **Thread component**: represents a schedulable and concurrent unit of sequentially executed source code.
- **Thread group component**: represents a single reference to a group of threads that have common characteristics and does not represent a unit of execution.
- **Subprogram component**: represents a callable piece of sequentially executed source code that operates on data or provides functions to the component that calls it.
- **Data component**: represents a data type to type port and subprogram parameter interfaces, and static data shareable among components.

**Execution platform components**

- **Processor component**: represents hardware with associated software that schedules and executes threads.
- **Virtual processor component**: represents a logical resource that is able to schedule and execute threads.
- **Memory component**: represents a storage for executable code and data.
- **Bus component**: represents a component that can exchange control and data between processors, memories, and devices.
- **Virtual bus component**: represents a logical bus abstraction.
- **Device component**: represents a dedicated entity within the system, or an entity in or interfacing with, the external environment, such as GPS systems, counters, timers, sensors, and actuators.
Chapter 2. Background

Changes to the runtime architecture can be described by modes and transitions of modes, and behavior of components can be described by state transitions systems defined in the Behavioral Annex (BA) [34]. AADL defines component abstractions which are further divided into three groups:

- **Application software components**
  - **Process component**: represents a protected address space containing at least one thread.
  - **Thread component**: represents a schedulable and concurrent unit of sequentially executed source code.
  - **Thread group component**: represents a single reference to a group of threads that have common characteristics and does not represent a unit of execution.
  - **Subprogram component**: represents a callable piece of sequentially executed source code that operates on data or provides functions to the component that calls it.
  - **Data component**: represents a data type to type port and subprogram parameter interfaces, and static data shareable among components.

- **Execution platform components**
  - **Processor component**: represents hardware with associated software that schedules and executes threads.
  - **Virtual processor component**: represents a logical resource that is able to schedule and execute threads.
  - **Memory component**: represents a storage for executable code and data.
  - **Bus component**: represents a component that can exchange control and data between processors, memories, and devices.
  - **Virtual bus component**: represents a logical bus abstraction.
  - **Device component**: represents a dedicated entity within the system, or an entity in or interfacing with, the external environment, such as GPS systems, counters, timers, sensors, and actuators.

- **General composite components**
  - **System component**: represents a composition of software, hardware, and/or system components, where the software components can be allocated to the hardware components.

A component is modeled by a **component type** and a **component implementation**.

### 2.5.1 Component Type

A component type is declared with an unique identifier and specifies the external interfaces (features) of the component, externally visible properties, and explicit data and control flows between the external interfaces. Consequently, a component type can be viewed as a black-box. Interfaces are declared with identifiers in a *features* sub clause, and represent interaction points for the exchange of data and control to other components. The *features* sub clause may include four different AADL elements: *port*, *component access*, *subprogram*, and *parameter*.

**Ports** represent interaction points for directional exchange of data, events, or both. A port can either be declared as a *data* port, an *event* port, or an *event data* port. A data port communicates typed state data, such as sensor data streams, without queuing, where the connection between data ports can be declared as immediate (transmitted upon completion of a thread) or delayed (transmitted upon the deadline). An event port communicates events, such as triggers for dispatches of threads, triggers for mode switches and alarms, with queueing. An event data port communicates messages, i.e., data associated with events, with queuing. Event and event data ports may transmit output at any time during the execution of the component through a *Send* . *Output* service call. Ports are directional and can either be an *in* port, an *out* port, or an *in out* port. An *in* port denotes an input of a component, an *out* port denotes an output of a component, and an *in out* port denotes an input and output of a component. An example of an out data port is shown in Figure 2.4.

**Component access** declarations support modeling of interfaces for the access of static data shareable among components, and modeling of hardware components communicating through buses. Access declarations are named and can be declared with a *provides* or *requires* statement. A provides statement denotes that a component provides access to a data or bus component internal to it. A requires statement denotes that a component requires access to
a data or bus component external to it. Examples of requires bus access statements are shown in Figure 2.3. There are three component types in the Figure: a bus component named ARINC_629, a memory component named SDRAM, and a processor component named PENTIUM. The connectivity of the execution platform through the bus is declared by require bus access statements in the memory and processor component features. Both access statements require bus access to the ARINC_629 bus component and are identified as controller_cpu and controller_memory.

A **Subprogram** feature represents an entry point into source text that operates on the component it is a feature of. A subprogram as a feature of a data component can be used to represent methods of objects.

**Parameters** represent interaction points of a subprogram for the transmission of call (in parameter) and return (out parameter) data values. Parameters are similar to ports, however, they can only communicate data and be features of subprograms.

Property declarations may be included in a *properties* sub clause of a component type. In addition, property declarations can be associated to most AADL expressions. A property constrains the expression it is associated with, and in this case, as a sub clause of a component type, constrains the component type. Examples of other expressions that can be associated with property declarations are: component implementations, subcomponents, features, connections, flows, modes, mode transitions, subprogram calls, and packages. A property declaration consists of a name, a type, and a value. The name cor-

```plaintext
bus ARINC_629
end ARINC_629;

processor PENTIUM
  features
    controller_cpu: requires bus access ARINC_629;
  end PENTIUM;

memory SDRAM
  features
    controller_memory: requires bus access ARINC_629;
  end SDRAM;
```

Figure 2.3: Examples of requires bus access statements
Chapter 2. Background

Figure 2.3: Examples of requires bus access statements. There are three component types in the Figure: a bus component named ARINC 629, a memory component named SDRAM, and a processor component named PENTIUM. The connectivity of the execution platform through the bus is declared by require bus access statements in the memory and processor component features. Both access statements require bus access to the ARINC 629 bus component and are identified as controller cpu and controller memory.

A subprogram feature represents an entry point into source text that operates on the component it is a feature of. A subprogram as a feature of a data component can be used to represent methods of objects. Parameters represent interaction points of a subprogram for the transmission of call (in parameter) and return (out parameter) data values. Parameters are similar to ports, however, they can only communicate data and be features of subprograms.

Property declarations may be included in a properties sub clause of a component type. In addition, property declarations can be associated to most AADL expressions. A property constrains the expression it is associated with, and in this case, as a sub clause of a component type, constraints the component type. Examples of other expressions that can be associated with property declarations are: component implementations, subcomponents, features, connections, flows, modes, mode transitions, subprogram calls, and packages. A property declaration consists of a name, a type, and a value. The name corresponds to the identifier of the property. The property type specifies a set of values that is accepted for a given property, and each property must be assigned a value or a list of values. There are built-in predeclared properties in the language but creation of new properties is supported. An example of a property statement is shown in Figure 2.4. The (built-in) property name is Latency, the property type is time, and the property value is 10 ms. This property specifies that it takes ten milliseconds until an output is available at the output port when the throttle is adjusted.

2.5.2 Component Implementation

A component implementation declaration represents the internal structure in terms of subcomponents and their connections, flow sequences, properties, component modes and mode transitions. Component implementations are coupled with a component type by associating the component type identifier with the component implementation identifier. The component implementation can, therefore, be viewed as a white-box in contrast to its component type. A component type can have zero, one, or several component implementations. The component implementation subcomponents sub clause represents a component’s internal components. These internal components can themselves have subcomponents resulting in a hierarchy that eventually describes a whole system. Examples of subcomponent statements are shown in Figure 2.5. The figure presents a component implementation, identified as impl, of a component type, identified as ac_computer (component type ac_computer is not shown in the figure), in addition to the three component types: PENTIUM, SDRAM, and ARINC_629. The component implementation impl specifies the internal structure of a system component. The subcomponents sub clause specifies three subcomponents: SDRAM, ARINC_629, and PENTIUM, which are identified

```
device throttle
  features
    throttle_status: out data port throttle_type;
  flows
    flow1: flow source throttle_status {
      Latency => 10 Ms;
    }
  end throttle;
```

Figure 2.4: Example of a property statement of a data flow
in the component implementation as airplane_memory, airplane_bus, and airplane_processor respectively.

Components interact through interfaces declared in the features sub clause (sub clause of a component type). The connections between interfaces are explicitly declared within a component implementation sub clause connections. A connection have an identifier, descriptor, source and destination. There are three types of connections: port connections, component access connections and parameter connections. The descriptor specifies whether the connection is a port (data, event, or data event port), an access (bus or data access) or parameter connection. Port connections represent directional transfer of data and control between ports. Examples of data port connections are shown in Figure 2.6. The system implementation ac_application.impl consists of three subcomponents: airplane_control.impl, throttle, and engine. The types for these components are shown in the figure. The connection declarations state that: 1) there is a data port connection named C1 from out data port throttle_status of
device throttle
defeatures
    throttle_status: out data port throttle_type;
end throttle;

device engine
defeatures
    engine_setting: in data port engine_type;
end engine;

process airplane_control
defeatures
    throttle_status: in data port throttle_type;
    engine_setting: out data port engine_type;
end airplane_control;

system implementation ac_application.impl
subcomponents
    AC: process airplane_control.impl;
    THROTTLE: device throttle;
    ENGINE: device engine;
connections
    C1: data port THROTTLE.throttle_status -> AC.throttle_status;
    C2: data port AC.engine_setting -> ENGINE.engine_setting;
end ac_application.impl;

Figure 2.6: Examples of connection statements

device THROTTLE to in data port throttle_status of process AC; and 2) there is a data port connection named C2 from out data port engine_setting of process AC to in data port engine_setting of device ENGINE.

A component access connection represents the path from the component providing access to the component requiring access. Examples of bus access connections are shown in Figure 2.5. Within the system component ac_computer.impl, there are three subcomponents: a processor, a memory, and a bus component, where the processor and memory components require bus access to the bus component. The connections are declared in the connection sub clause and state that: 1) there is a bus access connection named C1 from airplane_bus (ARINC_629) to the requires bus access feature named controller_memory in airplane_memory (SDRAM) component; and 2) there is a bus access connection C2 from airplane_bus to the requires bus access feature named controller_cpu in airplane_processor (PENTIUM) component.
Parameter connections represent flows of data into and out of subprograms, and data flows through a sequence of subprogram calls. Parameter connections can be declared between subprogram parameters or between a data port and a subprogram parameter.

Components can be modeled with modes and mode transitions, representing different configurations of contained components, connections, and prop-

```plaintext
thread automatic
features
  in_data : in data port;
  out_data : out data port;
end automatic;

thread manual
features
  in_data : in data port;
  out_data : out data port;
end manual;

process control
features
  input : in data port;
  output : out data port;
  trigger : in event port;
end control;

process implementation control.impl
subcomponents
  AUT: thread automatic;
  MAN: thread manual;
connections
  C1: data port input -> MAN.in_data in modes (manualmode);
  C2: data port MAN.out_data -> output in modes (manualmode);
  C3: data port input -> AUT.in_data in modes (automode);
  C4: data port AUT.out_data -> output in modes (automode);

modes
  manualmode: initial mode ;
  automode: mode ;
  manualmode -[ trigger ]-> automode;
  automode -[ trigger ]-> manualmode;
end control.impl;
```

Figure 2.7: Example of a mode state machine

Parameter connections represent flows of data into and out of subprograms, and data flows through a sequence of subprogram calls. Parameter connections can be declared between subprogram parameters or between a data port and a subprogram parameter.

Components can be modeled with modes and mode transitions, representing different configurations of contained components, connections, and prop-
2.5 The Architecture Analysis and Design Language

Parameter connections represent flows of data into and out of subprograms, and data flows through a sequence of subprogram calls. Parameter connections can be declared between subprogram parameters or between a data port and a subprogram parameter.

Components can be modeled with modes and mode transitions, representing different configurations of contained components, connections, and property value associations. These alternative modes are modeled in a component implementation. The mode transition system of a component must contain at least two modes, where one mode must be declared as the initial mode. A mode transition is declared with two modes and an event that triggers transition from one to the other. Each mode must have an explicitly defined configuration. An example of a mode state machine abstraction is shown in Figure 2.7. The process component control has one in data port, one out data port, and one in event port. In the control implementation, there are two declared thread subcomponents (AUT and MAN) and two declared modes: manualmode, which is specified to be the initial mode, and automode. In manualmode the process implementation is configured to have connections C1 and C2 while in automode it is configured to have connections C3 and C4. Transition from manualmode to automode and vice versa occurs by the trigger in event port.

Subprogram components are not declared as subcomponents, they are accessed through call statements. Calls are declared in the component implementation calls sub clause. An example of a call statement is shown in Figure 2.8 (parameters and connections are left out in the figure). The figure consists of one thread example.impl that calls a subprogram tank.temp where the identifier of the call is get.temp.

```
thread example
end example;

thread implementation example.impl
calls
{
get_temp: subprogram tank.temp;
};
end example.impl;

subprogram tank
end tank;

subprogram implementation tank.temp
end tank.temp;
```

Figure 2.8: Example of a call statement
Chapter 3

Research Overview

3.1 Research Challenges and Goals

In this section, we describe the challenges of this thesis, and define a research goal for each challenge. There are five primary challenges that have to be addressed in this thesis:

Challenge 1:
The first challenge is to validate if AADL adequately supports the development of techniques for efficient and effective formal verification and model-based testing. There generally exist two overarching needs that an ADL should support: 1) to increase understandability and communication among stakeholders; and 2) to facilitate V&V, preferably through computer tools [35, 10]. Both needs are important and contribute to the assurance of quality. There exist a number of ADLs that are able to represent architectures of dependable embedded systems and variably support these different needs. Hence, the right choice of ADL is a critical first step towards automated methods for quality assurance. This research project builds on earlier work within the areas of specification readability and literate specification language development, where AADL was considered due to the advantageous qualities of MetaH [10] – the language which AADL is derived from. However, the qualities of AADL cannot be concluded based on the qualities of MetaH, and several new ADLs have emerged since the first release of the AADL standard AS5506 [33] in 2004. To achieve an acceptable confidence in the language, validation of the current AADL standard with respect to alternatives is necessary.
Chapter 3

Research Overview

3.1 Research Challenges and Goals

In this section, we describe the challenges of this thesis, and define a research goal for each challenge. There are five primary challenges that have to be addressed in this thesis:

Challenge 1: The first challenge is to validate if AADL adequately supports the development of techniques for efficient and effective formal verification and model-based testing. There generally exist two overarching needs that an ADL should support: 1) to increase understandability and communication among stakeholders; and 2) to facilitate V&V, preferably through computer tools [35, 10]. Both needs are important and contribute to the assurance of quality. There exist a number of ADLs that are able to represent architectures of dependable embedded systems and variably support these different needs. Hence, the right choice of ADL is a critical first step towards automated methods for quality assurance. This research project builds on earlier work within the areas of specification readability and literate specification language development, where AADL was considered due to the advantageous qualities of MetaH [10] – the language which AADL is derived from. However, the qualities of AADL cannot be concluded based on the qualities of MetaH, and several new ADLs have emerged since the first release of the AADL standard AS5506 [33] in 2004. To achieve an acceptable confidence in the language, validation of the current AADL standard with respect to alternatives is necessary.
• **Goal 1:** Validate the capability of AADL to facilitate understandability, communication, and V&V through a comparison with alternative ADLs. The goal is achieved by the contributions of Paper A.

**Challenge 2:** The second challenge is related to the fundamental problems of software analysis and testing, regarding the objective, the data selection, the coverage criteria, and the oracle creation [36]. First, it is crucial to thoroughly understand why the analysis or testing should be performed, which is answered by identifying the objective. The objective of analyzing AADL models is logically to avoid architecture design faults. The objective of testing the developed system with respect to the AADL model is logically to avoid a faulty implementation of the architecture design. These objectives must, however, be decomposed and concretized with respect to the types of architecture design decisions that can be modeled by AADL and the types of design faults an AADL model can exhibit. Second, the relevant data must be identified to understand what kind of samples of an AADL model that must be analyzed or tested, and how they are extracted (the data selection). Third, the quantity of samples that is necessary to analyze or test must be determined to know when to stop the analysis or testing (the coverage criteria). Finally, the expected result of the analysis or testing must be identified to be able to decide if a result is acceptable or not (the oracle problem). These aspects must be addressed formally in order to be suitable for model-checking and model-based testing.

• **Goal 2:** Develop a formal verification technique for: 1) automated analysis of AADL models through model-checking; and 2) testing of a developed system with respect to its AADL model through model-based testing. The verification technique should formally define the objective, the data selection, the coverage criteria, and the oracle creation. The goal is achieved by the contributions of Paper B.

**Challenge 3:** The third challenge is that an adaption of model-checking and model-based testing techniques to AADL requires AADL models to be executable and have formally defined semantics. AADL has a precise dynamic semantics, but the semantics is not formally defined or implemented. Consequently, the semantics of AADL must be formalized and implemented before model-checking and model-based testing techniques can be adapted. Note that this challenge must be solved together with challenge 2 to completely achieve goal 2. The complete AADL execution model consists of numerous different aspects of a run-time environment, such as synchronous and asynchronous interactions, recovery execution, scheduling of threads, etc. A complete formalization and implementation of the execution model is, therefore, too extensive
Goal 1: Validate the capability of AADL to facilitate understandability, communication, and V&V through a comparison with alternative ADLs. The goal is achieved by the contributions of Paper A.

Challenge 2: The second challenge is related to the fundamental problems of software analysis and testing, regarding the objective, the data selection, the coverage criteria, and the oracle creation [36]. First, it is crucial to thoroughly understand why the analysis or testing should be performed, which is answered by identifying the objective. The objective of analyzing AADL models is logically to avoid architecture design faults. The objective of testing the developed system with respect to the AADL model is logically to avoid a faulty implementation of the architecture design. These objectives must, however, be decomposed and concretized with respect to the types of architecture design decisions that can be modeled by AADL and the types of design faults an AADL model can exhibit. Second, the relevant data must be identified to understand what kind of samples of an AADL model that must be analyzed or tested, and how they are extracted (the data selection). Third, the quantity of samples that is necessary to analyze or test must be determined to know when to stop the analysis or testing (the coverage criteria). Finally, the expected result of the analysis or testing must be identified to be able to decide if a result is acceptable or not (the oracle problem). These aspects must be addressed formally in order to be suitable for model-checking and model-based testing.

Goal 2: Develop a formal verification technique for: 1) automated analysis of AADL models through model-checking; and 2) testing of a developed system with respect to its AADL model through model-based testing. The verification technique should formally define the objective, the data selection, the coverage criteria, and the oracle creation. The goal is achieved by the contributions of Paper B.

Challenge 3: The third challenge is that an adaption of model-checking and model-based testing techniques to AADL requires AADL models to be executable and have formally defined semantics. AADL has a precise dynamic semantics, but the semantics is not formally defined or implemented. Consequently, the semantics of AADL must be formalized and implemented before model-checking and model-based testing techniques can be adapted. Note that this challenge must be solved together with challenge 2 to completely achieve goal 2. The complete AADL execution model consists of numerous different aspects of a run-time environment, such as synchronous and asynchronous interactions, recovery execution, scheduling of threads, etc. A complete formalization and implementation of the execution model is, therefore, too extensive to be feasible in this thesis work. A subset of AADL tailored for the target systems must be chosen.

Goal 3: Define a formal and implemented semantics of a subset of AADL tailored for dependable embedded systems. The goal is achieved by the contributions of Paper C.

Challenge 4: The fourth challenge is the lack of scalability of formal methods in general [37]. Techniques based on an exhaustive state-space search typically run into the so called “state-space explosion problem” when applied to industrial-sized systems [38]. In practice, this corresponds to an excessive memory and time consumption. Hence, the practical applicability and scalability of the technique to industrial systems must be analyzed and validated.

Goal 4: Validate the practical applicability and scalability of the proposed technique by applying it to an industrial system. The goal is achieved by the contributions of Paper C.

Challenge 5: The fifth challenge addresses the cost of regression verification. AADL models are subjected to modifications and, thereby, also to regression verification. In order to develop an architecture-based verification technique that is cost-efficiently applicable in practise, the technique must be integrated with appropriate methods for regression verification.

Goal 5: Integrate the architecture-based verification technique with methods for regression verification. The goal is achieved by the contributions of Paper D.

3.2 Related Work

3.2.1 Evaluating ADLs

Garlan and Shaw [2] recognized the increasing need for architectural software design to cope with the increasing size of software systems. A system architecture as described by Garlan and Shaw is the collection of components and the interactions (connections) among them. In line with this view, Medvidovic and Taylor [10] specify a framework describing features, provided by an ideal ADL, to model the conceptual architecture of a software system. In order to distinguish an ADL from other notations, they define components, component interfaces, connectors, and architectural configurations to be the main features of an ADL. The framework is limited to comparisons and classifications of
software ADLs, thus, it can not be used to compare AADL with alternative ADLs that are able to represent software architectures as well as hardware architectures.

3.2.2 Architecture-based Verification

Bass et al. [4] define architectural properties as those distinguished by interfaces. Due to the properties of an architecture, the primary focus of evaluation at this level is the integration of components as described by Eickelmann and Richardson [3] in their work about architecture-based defect prevention and detection. They describe two goals of integration testing – ensuring consistency of component interfaces and ensuring that data- and control-interactions between components are correct. The idea of taking traditional data-flow and control-flow analysis criteria to the architectural-level has been proposed by Jin and Offut [39], where data-flow and control-flow properties through system architectures are defined. Based on these properties, they define general architecture-based verification criteria applicable to any ADL treating components and connectors as separate entities interconnected through their interfaces. The criteria require all possible bindings of architectural data-flow and control-flow properties to be fully exercised. These criteria are, however, not applicable to AADL since connectors are not treated as separate entities.

Muccini et al. [40] present an ADL-independent approach which uses a software architecture as a reference model for testing the conformance of the implementation with respect to the architecture. Their approach to software architecture conformance testing is based on graphs of labeled transition systems that capture architectural behaviors, which is used for generating tests that can be converted to code-level conformance tests. Muccini et al. [41] also explore how regression testing at the software-architecture level can systematically be used in order to reduce cost of retesting modified systems. The method used for regression testing is based on the comparison of graphs representing a system and its modified version such that only test cases in which modified nodes are executed.

Additional approaches of architecture-based testing have been proposed in [42, 39]. The test generation is based on architecture models expressed by ADLs. The dynamic semantics of these ADLs are however not based on any formal theories such as a state-based formalism, preventing the adaption of model-based testing techniques. The obstacle is solved through model transformations. In [42], the model-based testing technique is based on Petri net models which are transformed from Acme descriptions. In [39], the technique
is based on Behavioral Graphs (BGs) - an extension of Petri nets - transformed from Wright descriptions.

### 3.2.3 Formal Semantics for Formal Verification and Model-Based Testing

Several studies have proposed a formal semantics of AADL for formal verification through transformations to constructs in a state-based formalism. Each of the works presented below contributes with a formal semantics. However, they have disadvantages with respect to dependable embedded systems, where synchronous data communication and fixed-priority preemptive scheduling are the commonly used scheduling features, and where timing properties are critical. A mapping of AADL behavioral semantics into Petri Nets is presented by Renault et al. in [43], where the objective is to verify that the system is free from deadlocks and that data communications among threads are correct. The methodology does not preserve timing constraints of the AADL model due to the lack of timing expressiveness of the target language. Liu et al. [44] use Uppaal as the formal underpinning for schedulability verification. The translation does not consider preemptive scheduling or data communication. Chkouri et al. [45] define a translation to the BIP (Behavior Interaction Priority) language in a natural language to enable simulation and formal verification. Yang et al. in [46] define formal semantics of a synchronous subset of AADL in both TASM [47] and Timed Transition Systems (TTS). A definition of formal semantics in TTS enables proofs of semantic-preservation to be generated during the model transformation from AADL to TASM. The notion of scheduling protocols is not included in the transformation, and the semantics is restricted to non-preemptive scheduling. Berthomieu et al. [48] transform a fixed-priority scheduling semantics into TTS through an intermediate Fiacre model, however, only non-preemptive scheduling is considered.

### 3.3 Research Methodology

The overarching goal of this research project is to improve the development process of dependable embedded systems. Our approach focuses on extending the theoretical knowledge in the areas of architecture-based development, such as architecture design, ADLs, formal verification of architecture models, model-based testing from architecture models, and regression verification. A sub-goal is to embody the extensions in the form of computer tools. Tools
enable easy adoption into industrial practice, however, industry often requires tools to be validated based on empirical methods. Through collaboration with industrial partners, we qualitatively, and in some cases quantitatively when feasible, validate the efficiency and effectiveness (such as time and memory consumption) of developed techniques by realistic case studies. The main steps of our research methodology are as follows:

1. Conduct literature reviews and surveys to form new research goals.
2. Check the validity of new research goals through collaboration with domain experts and industrial partners. Invalid goals are either reworked or disregarded.
3. Develop solutions and implement them in the form of computer tools.
4. Quantitatively validate the efficiency and effectiveness of implemented solutions through industrial case studies.

3.4 Summary of Results

In this section we provide a summary of the results of our work in terms of the included papers and the contributions of each paper. A clarification of the corresponding contribution for each goal as well as the relevant paper for each contribution is presented in table 3.1.

3.4.1 Contribution 1 – A Comparison Framework

In order to accomplish Goal 1, we propose a framework in paper A to compare the extent of support ADLs provide developers to ensure quality of dependable and software-intensive embedded systems. An ADL should support understandability and communication among stakeholders, partly by providing a simple, possibly graphical, syntax and a well understood semantics. An ADL should also support the development of tools for analysis and V&V, partly by providing multiple perspectives and, preferably, a formal syntax and semantics [49]. Both aspects contribute to the correctness of system architectures, but are seldom simultaneously supported [10]. The framework describes favorable features of an ADL and has been developed by extending the framework defined by Medvidovic and Taylor [10]. Medvidovic and Taylor’s framework was developed solely for pure software ADLs, hence, it cannot be applied to
ADLs that are able to describe both software architectures and hardware architectures. The extensions constitute what the main modeling constructs of an ADL are abstractions of, which basically are: architectural elements of software, architectural elements of hardware and architectural elements of software deployed on hardware. In addition, since dependable and software-intensive embedded systems commonly operate in safety-critical, mission-critical and time-critical environments, the framework is extended with preferable features for dependability and timing modeling.

### 3.4.2 Contribution 2 – A comparison of AADL and EAST-ADL

In paper A we accomplish Goal 1 by comparing AADL with the Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) [50], an ADL widely used both in the automotive industry and the research community, to validate the qualities of AADL. Although EAST-ADL was developed specifically for automotive systems, whereas AADL targets a variety of types of embedded systems, its established usage in industry motivates a comparison of their common abilities. The languages are compared based on the extended framework.

### 3.4.3 Contribution 3 – An Architecture-Based Verification Technique

To accomplish Goal 2, we propose, in paper B, an architecture-based verification technique for dependable and software-intensive embedded systems modeled in AADL. The technique is based on the idea of evaluating the integration of components at the architecture-level by using traditional control and data flow analysis criteria, as proposed by Jin and Offut [39]. Through the adaption of model-checking approaches to AADL, we are able to provide an automated way of verifying the correctness (if requirements are modeled), completeness, and consistency of an AADL model. Through the adaption of model-based testing approaches to AADL, we are able to provide an automated way of verifying the conformance of an implementation to its AADL model.

The verification process is driven by the formally defined architecture-based verification criteria that define the verification objective, data selection and coverage requirement. Graphs of data and control flow interactions among architecture components are generated when the criteria are applied to an AADL model. We present algorithms to extract the possible paths through
the branches – representing the possible interactions among interfaces of components – and their specific constraints. Constraints are mainly derived from AADL property annotations. Properties vary from specific data types and data ranges to more complex constraints, such as maximum latencies and scheduling properties (such as deadlines and priorities of threads). Each path is considered as a verification sequence. A verification sequence is, thereby, a sequence of architectural elements and the constraints (and possibly requirements) of that sequence. The elements of the sequence are necessary to be exercised in compliance with the semantics of AADL, the order of the sequence, and the constraints of the sequence, to verify a particular control- and/or data-flow. The sequence can be exercised through model-checking and/or model-based testing.

Model-checking is used to verify consistency, completeness, and correctness if requirements are available. The model is consistent if the possible paths (verification sequences) do not contradict their specific constraints (e.g. data is interacted with correct data types, with values within the range of the data type, with a latency not exceeding the maximum limit, without reaching deadlines of threads, etc.). The model is complete if all the paths are able to be activated by the specified input classes and a path will be activated for every class of input. Correctness can only be determined if requirements are associated with the AADL model, or if property declarations are considered as requirements. The model is correct if no path exceeds any requirement declarations. Hence, the coverage criteria require all possible paths to be exercised.

End-to-end paths (typically sensor-to-actuator paths) contain information about the initial state of the system, the input or sequence of input needed to stimulate an execution of the system according to the expected path, and the expected output or sequence of outputs. Hence, each path can later be used to automatically generate test cases, to test a system implemented to satisfy the model. Moreover, the expected timing constraints between input and output are also contained in end-to-end paths. This information can be used to test if the implemented system satisfies the timing constraints.

3.4.4 Contribution 4 – A Formal Semantics of a Subset of AADL

In order to automate the proposed verification technique, and thereby reaching Goal 3, the generated paths must be executed in accordance to the dynamic semantics of the language. To achieve this, we propose, in paper C, a formal and implemented semantics of a subset of AADL through transformation rules...
to timed automata, the input language to the UPPAAL model-checker. A transformation to timed automata does also allow for the use of the UPPAAL model-checker to automatically perform model-checking and model-based testing. We have chosen a subset of AADL consisting of synchronous interaction with fixed-priority preemptive scheduling since these constructs are commonly used in dependable embedded systems.

3.4.5 Contribution 5 – Validation of the Verification Technique

In paper C, we accomplish Goal 4 by using a case study of a safety-critical fuel-level estimation system developed by Scania. The system consists of typical components in a dependable embedded system. Examples of components are sensors, actuators, Electric Control Units (ECUs) executing the application software, and a Controller Area Network (CAN) bus interconnecting the ECUs. The main functionality of the system is to estimate the fuel level in the vehicle tanks and present the level on the display located in the dashboard. In addition, the driver is warned if the fuel level is below a predefined value. By modeling the system in AADL, we are able to apply the technique to an AADL model corresponding to a complex industrial system. Through different metrics, the relative amount of memory and time consumption with respect to the calculated design complexity can be measured to validate the scalability of the technique to industrial systems.

3.4.6 Contribution 6 – Regression Verification

In paper D, we accomplish Goal 5 by using the concept of specification slicing [51] through system dependence graphs (SDGs) [21]. We use this concept to determine the minimal subset of a regression verification suite (of verification sequences) necessary to retest a modified architecture. The concept is built on definitions of interdependencies among elements in a specification language such that the effects of a modification can be traced throughout the specification. Such interdependencies can be used to identify the parts of the AADL model that are not affected by a modification and are unnecessary to re-verify. Our approach is to define the control and data dependencies among expressions of an AADL model such that a SDG can be extracted from any AADL model. Through comparisons of SDGs extracted from an AADL model and its modified version, the modification can be identified and the effects of the modification can be traced throughout the SDG of the modified AADL model. Previous
verification sequences which do not cover modified or affected nodes can then be disregarded to generate a more efficient regression verification suite.

<table>
<thead>
<tr>
<th>Research goal</th>
<th>Contribution</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,2</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>D</td>
</tr>
</tbody>
</table>

Table 3.1: Relations between goals, contributions and paper A to D.
Chapter 4

Conclusion

In this licentiate thesis, I have shown the importance of architecture-based development processes of dependable embedded systems. ADLs are described as being a key solution to support developers with these processes, where this thesis is focused on AADL. A comparison framework describing an idealistic ADL for dependable embedded systems is initially defined in Chapter 5. The framework is developed to evaluate the degree AADL supports system architecture engineering. Results show that AADL satisfactorily supports the understanding and communication of dependable embedded system architectures. In addition, results show that AADL satisfactorily supports development of tools for V&V, partly by a precise semantics, heterogeneous modeling capabilities, and by the ability to assign constraints and requirements on architectural elements. This motivates the development of an architecture-based verification technique described in Chapter 6. The technique covers the entire development process, from the architecture design decisions to the architecture of the developed system. It is developed by adapting traditional formal methods, such as model-checking, and extensions thereof, such as model-based testing, to AADL. The use of formal methods is motivated by the increasing need for automated verification and evidences based on well-founded mathematics in the development of dependable embedded systems. The objective and verification criteria of the technique originate from the extensively researched areas of control-flow analysis, data-flow analysis and integration testing of software systems. We show that corresponding flows, which we call relations and mainly concern interactions of components, exist in AADL models. Furthermore, we describe how they can be used to verify the correctness, completeness
and consistency of AADL models, and the conformance of the architecture of the developed system with respect to its AADL model.

A definition of AADL-semantics in a formal domain is necessary to be able to complete an adaption of formal methods to AADL. We defined a formal semantics for a subset of AADL in Chapter 7. The subset is defined by using the concept of semantic anchoring, where the semantics is mapped to the formal domain of timed automata theory. The subset is chosen based on the common architectural patterns of dependable embedded systems. In addition, a mapping to timed automata allows for the use of well-founded UPPAAL technologies for model-checking and model-based testing.

Since model-checking and model-based testing involve exhaustive searches of system states, the verification activities themselves can be time-consuming although they are automated in UPPAAL. In addition, a development process of a dependable embedded system is likely to involve numerous modifications to the architecture design, analysis of closely related architecture design variants, and analysis of architectural trade-offs. Consequently, the process of verification is likely to be iterated in the development process, i.e., it is likely to be used for regression verification and regression testing. To avoid any bottle-necks in terms of time and memory consumption, we developed an algorithm presented in Chapter 8, to selectively extract only the architectural control- and data-flows that are necessary to exercise with respect to a regression verification or a regression testing process. The selection algorithm is based on traditional program and specification slicing algorithms, and it can be automated as it is mathematically founded.

4.1 Future Work

The techniques and algorithms proposed in this thesis are only formally defined, and not implemented in a computer tool. Unnecessary manual labor is, therefore, required when they are used although some processes of the architecture-based verification technique are automated in UPPAAL. A comprehensive future work task, involving the implementation of proposed techniques and algorithms in computer tools, is therefore possible. First, a tool for transformation of AADL-semantics to the formal domain of timed automata theory, as described in Chapter 7, could be developed. Existing editors for both languages supports XML (Extensible Markup Language) as an intermediate language, which can be used to easily implement the transformation rules. Second, the objective, sample (control and data flows) selection, and cover-
age criteria defined by the architecture-based verification technique presented in Chapter 6 has not been implemented yet. Without any computer tool, the extraction of verification samples according to the coverage criteria, and verification of the extracted samples according to the verification objective, must be handled manually in UPPAAL. Existing open source AADL editors, such as the OSATE (Open Source AADL Tool Environment) Eclipse IDE, could be extended to easily implement the extraction of verification samples. Samples of an AADL model can be mapped to the corresponding timed automata samples through the intermediate XML file. Constraints and requirements of AADL control and data flows can be transformed to UPPAAL logic formulae (query) files [27], and the addition of an observer automaton [52] for each flow, to the intermediate XML file. Third, an AADL version management database could be developed and integrated with the AADL-slicing algorithm defined in Chapter 8, to handle selective regression verification and testing.

In previous work [16, 53], we have shown the increasing need for evidences based on formal verification. In [16], we present an industrial case study investigating the risk of legal liability when constructing safety-critical software-intensive and mechatronic systems. In [53], we present an industrial case study describing experiences of constructing a safety case for safety certification of a safety-critical embedded automotive system. Further work includes an investigation of how the proposed verification techniques and algorithms in this thesis could be used to strengthen the safety argumentation and evidences when certifying dependable embedded systems. Furthermore, quantitative assessments of dependability attributes, such as reliability, availability, and safety, are important evidences in the certification of dependable embedded systems. Formal verification is a fundamental approach to mathematically analyze such properties. The Error Model annex is an extension of AADL that allows for stochastic modeling of errors and error propagation. As the occurrence probability of errors of flows can be specified, we believe that the verification technique proposed in this thesis can be used to analyze probabilistic failures. Through such analysis, quantitative assessments of reliability, availability, safety, etc., can be established.


[3] Nancy S. Eickelmann and Debra J. Richardson. What makes one software architecture more testable than another? In ISAW '96: 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 65–67, New York, NY , USA, 1996. ACM.


Bibliography


[3] Nancy S. Eickelmann and Debra J. Richardson. What makes one software architecture more testable than another? In ISAW ’96: 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 65–67, New York, NY, USA, 1996. ACM.


