The main goal of this doctoral thesis is to improve verification processes of dependable embedded systems. An embedded system is a computer system that has a dedicated function within a larger electrical and possibly mechanical system. A dependable embedded system is a computer system that is critical to the system it is embedded within. Examples of dependable embedded systems are electronic control systems in airplanes, trains, and cars, such as an autopilot. Since an incorrect operation of these systems may endanger people and the environment, it is crucial to verify that the systems achieve high quality before they are put into service. In this regard, verification means efforts that intend to detect and correct defects in the systems that are being developed.

Verification of embedded systems commonly involves manual work, which is becoming increasingly labor-intensive and error-prone due to the increasing complexity of the systems. The contribution of this doctoral thesis is a framework of verification techniques that provides an automated verification process to reduce the cost of human labor and the risk of human error. A comprehensive set of verification techniques is included in the framework so that defects are detected throughout the development process, including early design faults, intermediate implementation faults, and later maintenance faults. In addition, the framework is based on a mathematical foundation to ensure reliable verification results. Altogether, the framework provides a significant protection against costly and hazardous defects that may arise in the development of dependable embedded systems.

The performance of the verification framework is evaluated in this thesis by means of case studies, where, for example, it is applied to a safety-critical train control system. The results of the studies demonstrate a high fault detection rate and a scalability to advanced embedded systems with multi-core processors and multi-tasking.
QUALITY ASSURANCE FOR DEPENDABLE EMBEDDED SYSTEMS

Andreas Johnsen

2018

School of Innovation, Design and Engineering
QUALITY ASSURANCE FOR DEPENDABLE EMBEDDED SYSTEMS

Andreas Johnsen

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 26 januari 2018, 13.00 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor Paola Inverardi, University of L’Aquila

MÅLARDALEN UNIVERSITY SWEDEN

Akademin för innovation, design och teknik
Abstract

Architectural engineering of embedded computer systems comprehensively affects both the development processes and the abilities of the systems. Rigorous and holistic verification of architectural engineering is consequently essential in the development of safety-critical and mission-critical embedded systems, such as computer systems within aviation, automotive, and railway transportation, where even minor architectural defects may cause substantial cost and devastating harm. The increasing complexity of embedded systems renders this challenge unmanageable without the support of automated methods of verification, to reduce the cost of labor and the risk of human error.

The contribution of this thesis is an Architecture Quality Assurance Framework (AQAF) and a corresponding tool support, the Architecture Quality Assurance Tool (AQAT). AQAF provides a rigorous, holistic, and automated solution to the verification of critical embedded systems architectural engineering, from requirements analysis and design to implementation and maintenance. A rigorous and automated verification across the development process is achieved through the adaption and integration of formal methods to architectural engineering. The framework includes an architectural model checking technique for the detection of design faults, an architectural model-based test suite generation technique for the detection of implementation faults, and an architectural selective regression verification technique for an efficient detection of faults introduced by maintenance modifications.

An integrated solution provides traceability and coherency between the verification processes and the different artifacts under analysis, which is essential for obtaining reliable results, for meeting certification provisions, and for performing impact analyses of maintenance modifications. The Architecture Quality Assurance Tool (AQAT) implements the theory of AQAF and enables an effortless adoption into industrial practices. Empirical results from an industrial study present a high fault detection rate at both the design level and the implementation level as well as an efficient selective regression verification process. Furthermore, the results of a scalability evaluation show that the solution is scalable to complex many-core embedded systems with multithreading.

ISSN 1651-4238
Abstract

Architectural engineering of embedded computer systems comprehensively affects both the development processes and the abilities of the systems. Rigorous and holistic verification of architectural engineering is therefore essential in the development of safety-critical and mission-critical embedded systems, such as computer systems within aviation, automotive and railway transportation, where even minor architectural defects may cause substantial cost and devastating harm. The increasing complexity of embedded systems renders this challenge unmanageable without the support of automated methods of verification, to reduce the cost of labor and the risk of human error.

The contribution of this thesis is an Architecture Quality Assurance Framework (AQAF) and a corresponding tool support, the Architecture Quality Assurance Tool (AQAT). AQAF provides a rigorous, holistic and automated solution to the verification of architectural engineering of critical embedded systems, from requirements analysis and design to implementation and maintenance. A rigorous and automated verification across the development process is achieved through the adaption and integration of formal methods to architectural engineering. The framework includes an architectural model checking technique for the detection of design faults; an architectural model-based test suite generation technique for the detection of implementation faults; and an architectural selective regression verification technique for an efficient detection of faults introduced by maintenance modifications.

An integrated solution provides traceability and coherency between the verification processes and the different artifacts under analysis, which is essential for obtaining reliable results; for meeting certification provisions; and for performing impact analyses of maintenance modifications. The Architecture Quality Assurance Tool (AQAT) implements the theory of AQAF and enables an effortless adoption into industrial practices. Empirical results from an industrial study present a high fault detection rate at both the design level and the
implementation level, as well as an efficient selective regression verification process. Furthermore, the results of a scalability evaluation show that the solution is scalable to complex multi-core embedded systems with multi-tasking.
Swedish Popular Science

Summary

Denna doktorsavhandling handlar huvudsakligen om att förbättra verifieringsprocesser av tillförlitliga inbyggda system. Ett inbyggt system är ett datorsystem som har en specifik funktion inom ett större elektroniskt och möjligtvis mekaniskt system. I motsats till inbyggda system finns exempelvis persondata, som är utvecklade för ett stort antal olika användningsområden. Ett tillförlitligt inbyggt system är ett datorsystem vars funktion även är kritiskt för systemet det är inbyggt i och dess omgivning. Exempel på tillförlitliga inbyggda system är elektroniska styrsystem i flygplan, tåg och bilar, så som bilens farthållare. Eftersom en felaktig funktion av dessa system kan leda till fara för människor och omgivningen, är det avgörande att verifiera att systemen uppnår en hög kvalitet innan de distribueras och tas i bruk. Verifiering betyder i detta avseende medel som har för avsikt att upptäcka och korrigera defekter i systemet som utvecklas.

Verifiering av inbyggda system är en ständig växande utmaning som grundar sig i den ökande komplexiteten av systemen. Ända sidan gör den avancerade tekniken det möjligt att exempelvis utveckla effektiva transportmedel, så som autonom elbilar. När komplexiteten ökar måste systemarkitekter göra allt mer komplicerade beslut gällande val av arkitekturdesign. En arkitekturdesign är en övergripande teoretisk beskrivning av det system man planerar att utveckla. I beskrivningen ingår de väsentliga komponenterna, så som sensorer, processorer, ställdon, datorminnen, databussar, och applikationer, hur de skall vara strukturerade, och hur de skall interagera med varandra och med omgivningen systemet skall agera i. Med hänsyn till tillförlitliga system är arkitekturdesignen av exceptionell betydelse i den mening att graden av tillförlitlighet till stora delar bestäms av systemets struktur. Tillexempel innehåller fly-
gelektroniska system (avionik) ofta parallella datorer som utföra en och samma uppgift, så kallad redundans. Om den primära datorn fallerar kan systemet övergå till den sekundära och på så sätt bibehålla sin funktion och en säker flygning trots uppkomsten av ett allvarligt fel. Denna specifikation över arkitekturdesignen används sedan som underlag för den resterande utvecklingsprocessen, med intentionen att implementera ett system enligt specifikationen. Eftersom valet av arkitekturdesign bestämmer övergripande systemegenskaper och har långtgående påverkan på utvecklingsprocessen kommer ett felaktigt beslut sannolikt att resultera i ett system som inte uppnår kvalitetskraven, kostsamma korrigeringsarbeten, och i värsta fall personskador till följd av olyckor. Således är det nödvändigt att i största möjliga mån förhindra och upptäcka felaktiga beslut gällande systemets arkitekturdesign.

vilka delar av systemet som inte påverkas av en uppdatering och inte behöver ingå i omverifieringen är däremot en komplicerad uppgift. En förändring kan medföra både direkta och indirekta påverkningar, vilket gör den svåra att analysera med precision. Denna typ av omverifiering, där analyser utförs för att selektivt exekvera omverifieringsprocessen och på så sätt undgå onödvändiga omverifieringar, benämns som selektiv regressionsverifiering. En selektiv regressionsverifieringsteknik är resurseffektiv så länge tidskostnaden för analyser utförs för att selektivt exekvera omverifieringsprocessen och på så sätt undgå onödvändiga omverifieringar.

Bidraget av denna doktorsavhandling är en applikationsprogramvara, en så kallad “app”, som automatiskt och rigoröst utför verifieringar av arkitekturmodeller, modell-baserad testning av implementationer, samt selektiv regressionsverifiering av arkitektura arkitekturuppdateringar. Detta ger ett helhetligt skydd mot kostsamma och riskfyllda fel som uppstår i utvecklingsprocesser av tillförlitliga inbyggda system. För att uppnå pålitlighet har verktyget utvecklats med hjälp av formella verifieringsmetoder, vilka bygger på matematiska modeller och analyser.

Applikationsprogramvaran evalueras i denna avhandling genom en industriell fallstudie, vari verktyget tillämpas på ett säkerhetskritiskt inbyggt system. För att uppnå statistiskt signifikanta resultat och inkluderande av de olika feltyperna som potentiellt kan förekomma i en utvecklingsprocess, baseras studien på en felinjiceringsmetodik, vari ett stort antal defekter avsiktligt införs i systemet för att därefter undersöka om verktyget upptäcker dem. Resultatet av studien visar på en effektiv detektering av systemfel och en effektiv resursanvändning vid regressionsverifieringar. Utöver detta evalueras även verktygets skalbarhet i en studie bestående av ett flertal olika system, vad gäller både typ och komplexitet. Studien visar att verktyget är kapabelt till att verifiera komplexa datorsystem med multiprocessorer (“multi-core processors”) så väl som multikörning av processer (“multi-tasking”).
Acknowledgements

I would first of all like to express my deepest gratitude to my advisors, Prof. Kristina Lundqvist, Prof. Paul Pettersson and Dr. Kaj Hänninen, for their support, encouragement and guidance throughout my research. It has been nothing but a great pleasure to work with all of you.

There are so many other people to thank. I would like to jointly thank all the people at the IDT department, Mälardalen University. Many special thanks to: Prof. Lars Asplund who encouraged me to become a Ph.D. student; my office-mates over the years, Jiale Zhou, Abhilash Thekkilakattil, Hüseyin Aysan, Adnan Causevic and Etienne Borde, for all the help and great office hours; Mehrdad Saadatmand, Federico Ciccozzi, Antonio Cicchetti and Thomas Lev- eveque for all the fun that we have had both at work as well as outside of work; and professors Hans Hansson, Mikael Sjödin, Thomas Nolte, Cristina Secelleanu, Sasikumar Punnekkat, Ivica Crnkovic and Jan Carlson, for the valuable knowledge and feedback I have received over the years.

I would also like to thank Radu Dobrin, Daniel Sundmark, Frank Lüders, Andreas Gustavsson, Jan Gustafsson, Björn Lisper, Henrik Thane, Fredrik Ekstrand, Håkan Forsberg, Kristina Forsberg, Alfonso Pierantonio, Omar Jaradat, Mikael Åsberg, Aida Causevic, Leo Hatvani, Juraj Feljan, Aneta Vulgarakis, Moris Behnam, Saad Mubeen, Dag Nyström, Sèverine Sentilles, Damir Isovic, Mikael Ekström, Nikola Petrovic, Stephan Baumgart, Iain Bate, Yue Lu, Husni Khanfar, Guillermo Rodriguez-Navas, et al., for all the help, inspiration and interesting discussions. I thank the administrative staff, Jenny Hägglund, Malin Rosqvist, Carola Rytersson, Sofia Jäderen, Gunnar Widforss, Susanne Fronnå, et al., for making many things easier.

Finally, I would like to express my deepest gratitude to my beloved family. My dear son, Carl-Axel, I hope you one day will read this. You are always in my thoughts, forever in my heart.
List of Publications

Papers Included in the Thesis

**Paper A** An Architecture-based Verification Technique for AADL Specifications; Andreas Johnsen, Paul Pettersson, and Kristina Lundqvist; In proceedings of the 5th European Conference on Software Architecture (ECSA), Essen, Germany, 2011.

**Paper B** Automated Verification of AADL Specifications Using UPPAAL; Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, and Omar Jaradat; In proceedings of the 14th IEEE International Symposium on High Assurance Systems Engineering (HASE), Omaha, NE, USA, 2012.

**Paper C** Regression Verification of AADL Models through Slicing of System Dependence Graphs; Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, and Kaj Hänninen; In proceedings of the tenth International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA), Lille, France, 2014.

**Paper D** AQAF: an Architecture Quality Assurance Framework for systems modeled in AADL; Andreas Johnsen, Kristina Lundqvist, Kaj Hänninen, Paul Pettersson, and Martin Torelm; In proceedings of the 12th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA), Venice, Italy, 2016.


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

Related Papers


- Industrial Experiences of Building a Safety Case in Compliance with ISO 26262; Raghad Dardar, Barbara Gallina, Andreas Johnsen, Kristina Lundqvist, and Mattias Nyberg; In proceedings of the 2nd Workshop on Software Certification (WoSoCER), Dallas, TX, USA, 2012.


Licentiate Thesis

- Architecture-based Verification of Dependable Embedded Systems; Andreas Johnsen; Mälardalen University, Västerås, Sweden, 2013.
# Contents

## I Thesis

1 Introduction ............................................... 3
   1.1 Overview of Research Area .......................... 3
   1.2 Problem Formulation and Motivation .............. 6
   1.3 Contribution to Research Area ..................... 9
   1.4 Outline ............................................. 11

2 Background Information ................................. 13
   2.1 The Architecture Analysis and Design Language ... 13
       2.1.1 Sensor-To-Actuator AADL Example ........... 17
   2.2 Control and Data Flow Analysis .................. 23
   2.3 Model Checking .................................. 26
       2.3.1 Modeling ..................................... 26
       2.3.2 Requirements Specification and Verification .. 28
   2.4 Model-based Testing .............................. 32
   2.5 Regression Verification ........................... 35
   2.6 Program Slicing .................................. 35

3 Research Methodology ................................ 39
   3.1 Research Method .................................. 39
   3.2 Research Challenges and Goals ................... 41

4 Related Work ........................................... 45
   4.1 Verification Criteria ............................... 45
   4.2 Formal Verification ................................. 46
   4.3 Model-based Testing ............................... 47
   4.4 Regression Verification ............................ 48
4.5 Verification Framework ........................................ 49
4.6 Hazard Analysis .................................................. 50

5 Summary of Contributions ........................................... 53
  5.1 Architecture Flow Graphs ....................................... 55
  5.2 Verification Criteria ........................................... 58
    5.2.1 Extension of Paper A: Inclusion of Data Flow Sources
          and Data Flow Sinks ........................................ 59
  5.3 Formal Semantics .............................................. 60
    5.3.1 Extension of Paper B: Inclusion of Shared Resources ... 62
  5.4 Model Checking with Observer Automata ....................... 69
  5.5 Model-based Testing with Observer Automata .................. 70
  5.6 Selective Regression Verification ............................. 70
    5.6.1 Extension of Paper D: Dynamic Dependence Analysis ... 72
  5.7 The Architecture Quality Assurance Tool ....................... 75

6 Evaluation ......................................................... 87
  6.1 Utility .......................................................... 87
    6.1.1 Design of Study ........................................... 87
    6.1.2 Results .................................................... 89
    6.1.3 Discussion ................................................. 92
  6.2 Scalability ...................................................... 92
    6.2.1 Design of Study ........................................... 93
    6.2.2 Results .................................................... 95
    6.2.3 Discussion ................................................. 105

7 Limitations And Future Work ...................................... 107

8 Conclusion .......................................................... 111

Bibliography .......................................................... 113

II Included Papers ................................................... 123

9 Paper A
   An Architecture-based Verification Technique for AADL Specifi-
   cations .......................................................... 125
   9.1 Introduction ................................................... 127
   9.2 Preliminaries .................................................. 128
9.3 The Architecture-based Verification Technique . . . . . . . . 128
9.4 AADL Verification Criteria . . . . . . . . . . . . . . . . . 130
  9.4.1 Verification objectives . . . . . . . . . . . . . . . . 130
  9.4.2 Verification Criteria . . . . . . . . . . . . . . . . . . 132
9.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . 135
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

10 Paper B
Automated Verification of AADL-Specifications Using UPPAAL  
10.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 141
10.2 AADL and The Verification Technique . . . . . . . . . . . . 142
10.3 Transformation to Timed Automata . . . . . . . . . . . . . 146
  10.3.1 Transformation Rules . . . . . . . . . . . . . . . . . 148
10.4 Case study . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
10.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . 158
10.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159

11 Paper C
Regression Verification of AADL Models through Slicing of System 
Dependence Graphs  
11.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 163
11.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
11.3 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
11.4 Slicing through system dependence graphs . . . . . . . . . . 170
  11.4.1 Generating Control Flow Graphs . . . . . . . . . . . . 172
  11.4.2 Generating Program Dependence Graphs . . . . . . . 178
  11.4.3 Generating the System Dependence Graph . . . . . . 180
11.5 Slicing and Selection . . . . . . . . . . . . . . . . . . . . . . 185
11.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187

12 Paper D
AQAF: an Architecture Quality Assurance Framework for systems 
modeled in AADL  
12.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 191
12.2 Line Trip Relay Interface and Supervision . . . . . . . . . . . 193
12.3 Architecture Flow Graphs . . . . . . . . . . . . . . . . . . . 196
  12.3.1 Architecture Flow Graph Generation . . . . . . . . . . 198
### 12.4 Verification Criteria and Sequences

- Verification Criteria and Sequences: 201
- Formal Semantics in Timed Automata: 203
- Observers Generation and Model Checking: 205
- Model-based Testing: 208
- Selective Regression Verification: 209
- Validation: 211
- Related Work: 214
- Conclusion: 215
- Bibliography: 217

### 13 Paper E

**AQAT: The Architecture Quality Assurance Tool for Critical Embedded Systems**

- Introduction: 221
- AQAT Front-end and Usage: 222
  - Use Cases: 222
  - User Interface: 225
- AQAT Back-End: 231
  - Architecture Analysis and Design Language: 231
  - Architecture Flow Graphs: 234
  - Verification Criteria and Verification Sequences: 237
  - Transformation To Timed Automata: 238
  - Model Checking: 240
  - Test Suite Generation: 242
  - Selective Regression Verification: 243
- Evaluation: 245
- Related Work: 246
- Conclusion and Future Work: 247
- Bibliography: 249

### 14 Paper F


- Introduction: 255
- The Architecture Quality Assurance Framework: 256
- Case Study Design: 259
  - Objectives of Study: 259
  - Case Selection: 259
- Bibliography: 249
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.3.3</td>
<td>The Case Study Subject</td>
<td>260</td>
</tr>
<tr>
<td>14.3.4</td>
<td>Case Study Method and Expected Results</td>
<td>262</td>
</tr>
<tr>
<td>14.3.5</td>
<td>Fault Types and Fault Injections</td>
<td>265</td>
</tr>
<tr>
<td>14.3.6</td>
<td>Case Study Instrumentation</td>
<td>266</td>
</tr>
<tr>
<td>14.4</td>
<td>Results</td>
<td>268</td>
</tr>
<tr>
<td>14.5</td>
<td>Discussion</td>
<td>276</td>
</tr>
<tr>
<td>14.6</td>
<td>Related work</td>
<td>278</td>
</tr>
<tr>
<td>14.7</td>
<td>Conclusion and Future Work</td>
<td>279</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>281</td>
</tr>
</tbody>
</table>
I

Thesis
Chapter 1

Introduction

1.1 Overview of Research Area

Digital technology is an integral part of societies and often used in applications where the welfare and the safety of humans and environments are dependent on it [1]. For example, computer systems are embedded in control systems of nuclear power plants, public transportation, railway vehicles, automobiles, aircraft, and medical equipment. These types of computer systems, which have a dedicated function within a larger electrical and possibly mechanical system, are referred to as embedded systems [2]. Dependability of computer systems is a generic term including the attributes of availability, reliability, safety, integrity, and maintainability [3]. A dependable embedded system is an embedded system that provides a combination of these attributes. Embedded systems which may endanger the environment and humans are referred to as safety-critical or safety-related [4]. Safety-critical systems are generally developed under the requirement of functional safety: the absence of unacceptable/unreasonable risk due to hazards caused by malfunctioning behavior of the system [4]. Governments typically enforce functional safety standards in domains where the consequences of malfunction may be severe or catastrophic, to control that citizens and the environment are not exposed to an unacceptably high risk of harm [5][6]. Safety-critical systems are certified as safe when they have been justifiably demonstrated and proven, by the standardized measures, to not cause hazardous failures that are more frequent and more severe than acceptable [4]. This is an effortful challenge as functional safety standards do not only demand extensive safety and quality evidences of highly complex
Chapter 1. Introduction

products, but also evidences of an appropriately applied development process, which themselves are extensive for safety-critical systems [7]. In this thesis, quality of a system is defined in terms of defect density: the amount of defects (also referred to as “faults”) in the system with respect to its size [8]. In other words, quality is the degree to which a system complies with the expectations of it. In order for such a notion to be practicable, the expectations must be documented in terms of requirements. In this manner, a defect is defined as a condition within the system that violates, or may cause the system to violate, one or several requirements.

An embedded system which functionality is an absolute necessity to carry out the mission of the system it is embedded within, but is not necessarily safety-critical, is commonly referred to as a mission-critical system [9]. Although mission-critical systems that are not safety-critical do not possess an immediate threat to their environments, malfunctions of such systems typically have serious impacts on business operations and are therefore developed under stringent requirements of reliability and availability. Embedded systems are also commonly performance-critical and subjected to real-time constraints: services that are not provided in compliance with timing requirements are considered as system failures [10][11]. Anti-lock braking systems (ABS) and airbag control units are common examples of highly performance-critical systems. For example, the time of airbag inflation must be triggered within a period of milliseconds after a collision to function safely.

The lifecycle of embedded systems generally includes a set of consecutive stages: requirements analysis, design, implementation, integration, production, operation, maintenance, and finally decommissioning. For critical systems, the development process is complemented with hazard analysis and risk assessment and classification processes [4]. The different stages may be carried out with a number of iterations and an amount of overlaps between them. Requirements analysis is performed to understand the needs of stakeholders, such as customers, users, managers, certification authorities, investors, developers, and suppliers. The creation of the architectural design, which is a structural representation of the system intended to be produced, is the very first step of ensuring that the end product will meet the extra-functional requirements (also known as non-functional requirements and quality attributes) in addition to the functional requirements [12]. Functional requirements correspond to the expected responses of a system with respect to a set of possible inputs, such as the release of an airbag in response to a collision. Extra-functional requirements correspond to the expected criteria under which the functionality should operate, such as an airbag that operates safely and timely and is available for
use at all times despite long periods of inaction. The task of the architectural design process is essentially to allocate the required functionality to particular structures of components and connections that are known to exhibit certain extra-functional properties, such that the requirements of safety, performance, availability, reliability, modifiability, testability, security, etc., are achieved. For dependable embedded systems, the process includes allocation of functions to structures of hardware, software, information, and time redundancy to construct mechanisms of fault tolerance. For example, fly-by-wire systems are commonly composed of multiple redundant computers, sensors, and actuators that are engaged when the primary components fail, to maintain a safe flight despite the presence of errors. The architectural design is subsequently used as a blueprint among stakeholders and serves as a basis for the development process, where each component and connection of the design will be refined until an implementation (product) can be created based on the design.

In essence, the threat to dependability is faults [3]. A fault can take various forms, ranging from hazard analysis and development faults, such as exclusions of plausible hazardous malfunctions, incorrect design decisions, and software bugs, to operational and maintenance faults, such as random hardware failures and erroneous system upgrades. The avoidance of faults is essential for the achievement of dependability [13]. Fault avoidance is a term used to represent methods focused on developing a system that is fault-free. It includes both proactive methods, referred to as fault prevention, which prevent faults from being introduced in the development process, and reactive methods, referred to as fault removal, which detect and remove those faults that nevertheless are introduced. Although the development of a completely fault-free system is not feasible in practice and has to be supported with methods of fault tolerance, the use of fault avoidance is an absolute necessity to assure dependability since fault tolerance mechanisms themselves must be developed with fault avoidance to be sufficiently dependable.

The avoidance of faults is achieved through a systematic development process with continuous verification and validation (V&V) activities. Methods of validation ensure that the right system is built whereas methods of verification ensure that the system is built right [14]. Validation often involves judgments from stakeholders, customers, and users, and typically includes examination of high-level system and user requirements and demonstrations of compliance to those requirements. Verification is typically performed against a documented specification of expectations by three different approaches: (i) reason about and analyze the system through abstractions, such as simulation, data-flow analysis, model checking, theorem proving, abstract interpretation, etc.; (ii) inspect
1.2 Problem Formulation and Motivation

The complexity of critical embedded systems is increasing beyond what current engineering practices are able to manage [15][16][17]. A problem is that failures emerge in the interactions of components when the complexity increases whereas failures of individual components have received most focus in previous research [15]. Component interaction is a central concern of architectural design, which together with requirements analysis typically account for the majority of faults introduced in the development process [18]. In addition, the main objective of designing the architecture is to build necessary extra-functional properties, such as safety, reliability, and availability, into the system. An architectural design fault may consequently not only cause a failure of the critical functionality, but also of built-in fault tolerance mechanisms that are supposed to maintain dependability in the presence of errors.

The effects of a faulty architectural design also have a significant impact on business goals. In [19], a survey quantitatively evaluates the return on investment of system engineering based on an analysis of 161 projects. Results 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. The cost of finding and correcting faults generally increases with the progress of the development process from the time of introduction [20]. Defects in the architectural design are of particular criticality as they are created early and have a comprehensive impact on both the development process and the system. In [21], a survey of projects executed by defense contractors quantifies the relationship between system engineering practices and the performance of the projects in terms of cost, schedule, and scope goals. The results suggest that there is a strong positive relationship between architectural engineering capabilities and the performance of the projects. For example, only 11% of the projects with lower capabilities exhibited a good performance compared to 46% of the projects with higher capabilities. Safer and more dependable systems can consequently be developed at lower costs by improved architectural engineering.

The main solution to costly corrections and hazardous systems due to faults is to prevent them by a systematic and rigorous development process and to detect and remove those that have not been prevented as early as possible.
by verification. The need for advancement in architectural engineering due to increasing system complexities has, in line with the progress of model-driven engineering [22], led to the development of architecture description languages (ADLs). Model-driven engineering abstracts systems to standardized and computer-readable models that enable rigorous analysis in the early phases of development processes [16][17]. Medvidovic and Taylor [23] define an ADL as a textual and/or graphical language that can represent the components of a system, the interfaces of components through which the components interact, and the connections between the interfaces. The usage of ADLs in the development of embedded systems consequently generates computer-readable models of the architectural designs, which in turn enables development of computerized verification tools that are effective and efficient enough in detecting architectural faults that emerge in complex embedded systems.

The chronologically first challenge of architectural verification is to ensure, as much as reasonably practicable, that the architectural design is free from faults [12]. Nevertheless, a fault-free design does not imply a corresponding implementation as faults may be introduced in the process of implementing the design. Since achievements of critical requirements are dependent on the design, it is crucial to test that the implementation conforms to the design. The system may otherwise possess unanticipated properties that violates critical requirements. Such faults are primarily detected through testing, which is the second challenge of architectural verification.

Testing of critical software systems often consumes a majority of the total development cost, where the use of formal notations for architectural design has the potential to significantly reduce integration testing costs through automated test case generation [24]. Furthermore, the system lifecycle typically includes modifications to the architectural design due to maintenance, product line and variability development, and tradeoff analysis. Modifications require reverification measures since they may induce erroneous behaviors to previously functioning architectural elements. This type of reverification is collectively denoted as regression verification in this thesis, which generalizes the concepts and principles of the term regression testing to any type of verification performed in response to a change. Regression verification of the complete artifacts is inefficient if the change does not have the corresponding extensive effect. Regression testing, alone, consumes up to one-third of the total development cost of a software system [25]. A problem of regression verification in architectural engineering is that the impact of a modification on the remaining architecture is difficult to analyze with precision, as changes indirectly may negatively affect remote components and connections [26]. The use of ADLs
provides the ability to perform automatic and rigorous change impact analysis such that regression verification can be efficiently performed, where only those component and connections that may be impacted by the change are reverified [25], known as selective regression verification. A selective regression verification technique is efficient only if the overhead of conducting the impact analysis and selection does not exceed the gain of reducing the scope of a re-run all approach.

Verification of architectural engineering must consequently be holistically managed, from requirements analysis and design to implementation and maintenance. State of the art architectural verification techniques integrate formal verification methods with ADLs. The incentive for using formal methods is threefold. First, verification based on computer mathematics is rigorous and reliable [27]. Second, formal methods enable automation through computer tools, thereby reducing the cost of labor and the risk of human error [28]. Third, in our questionnaire [5] involving nine respondents from six international companies developing safety-critical embedded systems in the vehicular and avionics domains, results suggest that formal verification is ranked as one of the most important types of verification to mitigate liability issues. Nevertheless, only 43% of the respondents replied that formal verification is used, suggesting that there is a need for solutions that are more usable and easily adoptable for industry than currently available. A common problem of formal verification techniques is scalability to industrial-sized systems. Formal verification techniques commonly include exhaustive analyses of the system state spaces, which tend to increase exponentially when complexity is added to the systems. The phenomenon is known as the state explosion problem [29] and critical to overcome to develop solutions that are of use for industry.

To summarize, the problems addressed in this thesis originates from the increasing complexity of critical embedded systems [15][16][17], the importance of architectural engineering and verification thereof in the development processes [18][19][20][21], and the limited previous research on, and methods for, faults that emerge in the interactions of components [15]. Research incentives suggest that the progress of model-driven engineering, including the development of ADLs [23], may be used to improve verification of architectural engineering, in particular verification of the design [16][17], of the implementation [24], and of maintenance modifications[25]. These challenges constitute the domain in which the research and development presented in this thesis intend to provide a solution. The subsequent problems are:

- The solution should be holistic, such that faults can be detected as early
1.3 Contribution to Research Area

as possible throughout the development process.

- The solution should be automated, to reduce the cost of labor and the risk of human error.
- The solution should be built upon formal methods, to ensure accurate and reliable results.
- Finally, the solution should be easily adoptable and of utility for industry and scalable to modern embedded systems.

1.3 Contribution to Research Area

The main contribution of this thesis is the Architecture Quality Assurance Framework (AQAF), providing a holistic, formal, and automated solution to architectural verification of critical embedded system, and a corresponding tool support – the Architecture Quality Assurance Tool (AQAT). AQAF includes an architectural model checking technique to address architectural design faults; an architectural model-based test suite generation technique to address architectural implementation faults; and an architectural selective regression verification technique to address faults introduced by architectural maintenance modifications. The verification objective of AQAF is to ensure executability of prescribed architectural control and data flows and compliance of those executions with respect to functional and extra-functional requirements. The contribution is of importance to industry as contemporary standards, such as the functional safety standard ISO 26262 [30] for automotive systems, require control and data flow analysis of architectural designs, formal verification of systems with high risk, tests that demonstrate conformance of an implementation with respect to its architectural design, and impact analysis of design changes to identify necessary regression verification measures.

Several ideas, techniques, and tools have previously been developed to formally and automatically perform various types of architectural verifications, as presented in Section 4. However, the majority of previous contributions address these challenges individually, without regard to their interconnections, making them difficult to combine in a development process. The verification techniques provided by AQAF are developed upon a common formal underpinning composed of the theories of timed automata and graphs. Graph theory is used to formally capture the prescribed control flows, data flows, and requirements of an architectural design, which also provides a basis on which
the impact of a design change can be analyzed through graph operations. Automata theory is used to anchor the semantics of an architectural design in a format appropriate for model checking and model-based test suite generation. Through the common formal underpinning, AQAF enables the method of performing model-based test suite generation based on the results of model checking [28]. The test oracle is thereby inherently consistent with the model-checked architectural behavior. Furthermore, a common formal underpinning provides explicit trace links between the graphs, the formal semantics, the verification runs, the coverage of the architectural design, and the coverage of the architectural implementation. Regression verification can thereby be efficiently executed by only selecting verification runs of the design and implementation that can be traced from the impact analysis.

AQAF is tailored for architectural engineering practices that use the Architecture Analysis and Design Language (AADL) [31] for the specification of the architectural design. AADL is selected as the reference architecture description language due to its ability to express properties that are essential for analyses of critical embedded systems and due to the thoroughly defined semantics [32] (a thoroughly defined semantics is an essential prerequisite for computer-aided verification). Although the framework is tailored for AADL in this thesis, the underlying theory is compatible with any other architecture description language that may express the targeted types of execution models. AQAF is primarily developed for architectures with synchronous, fixed-priority preemptive or non-preemptive execution models, as these commonly are used in critical embedded systems. Principles of modularization are however used to enable incorporation of other types of execution models.

In order to enable an effortless adoption into industrial practice, AQAF is implemented in a computer tool referred to as the Architecture Quality Assurance Tool (AQAT) – available for academic and non-commercial use at http://www.idt.mdh.se/~AQAT/. The utility of AQAT and the underlying AQAF theory are evaluated by means of an industrial case study, where AQAT is applied to a safety-critical train control system. Results suggest a statistically significant fault detection rate at both the design level and the implementation level, and an efficient selective regression verification technique. Furthermore, the scalability of AQAT is evaluated in a study with a wide range of system types and complexities. The results show that the tool is scalable to complex multi-core embedded systems with multi-tasking.
1.4 Outline

The thesis is formatted as a compilation thesis, also referred to as a thesis by publication, and is composed of two parts: an introductory part (part one) and a compilation of papers part (part two). The introductory part presents the research area, the research project, the related work, summaries of the contributions of the included papers, and the results, the limitations, and the potential future work of the presented contributions. More precisely, in chapter 2, the basics of AADL, control and data flow analysis, model checking, model-based testing, and regression verification are presented. The methodology of the research project is subsequently presented in chapter 3 together with a list of research challenges and goals. Summaries of the related work in the areas of architectural verification criteria, formal verification, model-based testing, regression verification, and hazard analysis are then presented in chapter 4, which are contrasted in chapter 5 with the thesis contributions, including references to the corresponding papers in part two. In chapter 6, the results of utility and scalability evaluations of the contributions are presented. The limitations of the contributions and potential future work are then presented in Section 7. Concluding remarks are finally presented in Section 8.
Chapter 2

Background Information

2.1 The Architecture Analysis and Design Language

AADL was initially released and published as a Society of Automotive Engineers (SAE) Standard (AS5506) in 2004 [33], and a second version was published in 2009 [31]. It is a textual and graphical language used to model, specify, and analyze software- and hardware-architectures of embedded systems. AADL is based on a component-connector paradigm that hierarchically describes an embedded system as a set of components, which themselves may be composed of (sub)components, a set of component interfaces, and a set of connections between interfaces through which components interact. Hence, the language captures functional properties of the system, such as input and output by component interfaces, as well as structural properties by configurations of components, subcomponents, and their connections. Means to describe extra-functional properties, such as timing and reliability, of application software and hardware platform components are also provided through property declarations. Furthermore, dynamic changes to the runtime configuration can be described by modes and mode transitions and the behavior of components can be modeled by state transitions systems defined by the AADL Behavioral Annex [34]. AADL defines component abstractions dividable into three groups:

- **Application software components**
  - **Process component**: represents a protected address space (must contain at least one thread).
– **Thread component**: represents a schedulable and concurrent unit of sequentially executed source code.

– **Subprogram component**: represents a callable piece of sequentially executed source code.

– **Data component**: represents a data type to type interfaces or a static data object possibly sharable among multiple components.

**Execution platform components**

– **Processor component**: represents hardware with associated software that schedules and executes threads.

– **Memory component**: represents a storage for executable code and data.

– **Bus component**: represents a component that transfers data between processors, memories, and devices.

– **Device component**: represents an electric or electronic entity that possibly interfaces with the external environment, such as sensors and actuators.

**General composite components**

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

A component is modeled by a component type declaration and a component implementation declaration, as presented in Backus-Naur Form in Table 2.1. A component type specifies the interfaces and the externally visible properties of the component. Interfaces are declared in a features subclause and represent points of interaction for the exchange of data and control to other components. Three different types of features (interfaces) can be modeled: ports, component accesses, and parameters. Ports are directional (in, out, or inout) and can be declared as a data port, an event port, or an event data port. A data port communicates state data without queuing, such as sensor data streams, where the connection between data ports can be declared as immediate (transmitted upon completion of the thread) or delayed (transmitted upon the deadline of the thread). An event port communicates events with queuing, such as dispatch triggers of threads, triggers for mode switches, and alarms. An event data port
2.1 The Architecture Analysis and Design Language

communicates messages, i.e., data associated with events, with queuing. Parameters exclusively represent interaction points of subprograms for the transmission of call (in parameter) and return (out parameter) data values. Component access declarations support modeling 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 component, bus component, memory component, etc., that is internal to it. A requires statement denotes that a component requires access to a data component, bus component, memory component, etc., that is external to it.

A component implementation declaration represents the internal component structure of a component type, in terms of subcomponents and their connections, component-internal properties, and modes and the transitions between modes. The component implementation subcomponents sub clause specifies the internal subcomponents of the component. These internal components can themselves have subcomponents resulting in a hierarchy that eventually describes the whole system. The connections between interfaces of the component and its subcomponents and between the subcomponents are declared within a connections sub clause of a component implementation. There are three types of connections: port connections, component access connections, and parameter connections. Port connections represent directional transfer of data and/or control between ports. A component access connection represents a link from the component providing access to a component to the component requiring access to it. Parameter connections represent flows of data into and out of subprogram components, which are invoked through call statements.

Property declarations may be added to virtually any element of the architectural design. A property constrains the expression it is associated with. For example, properties may be declared for a component type, such as the execution time, dispatch protocol, deadline, and priority of a thread. A property declaration consists of a name and a value. The name corresponds to the identifier of the property, which must be defined with a type that specifies a set of acceptable values for the given property. There exist built-in predeclared properties in the language, such as scheduling properties of threads, but creation of custom properties is supported.

Component implementations may also be modeled with mode state machines to specify the set of components, connections, and properties that are active in a specific mode, such as an engaged autopilot in aircraft or a disengaged driver assistance system within cars.
A mode state machine is composed of a set of modes and a set of mode transitions on the form $m \xrightarrow{\text{trigger}} m'$, where $m$ is the source mode, $m'$ is the target mode, and trigger is the event or event data port that triggers a transition of modes. A mode transition is engaged when the source mode is active and an
event arrives to the triggering port, whereupon the source mode is deactivated and the target mode is activated. Finally, a component implementations may be modeled with behavioral models (automata) of states and state transitions to model the logical execution behavior of the component. A behavior specification is composed of a set of states and a set of state transitions on the form 

\[ s \xrightarrow{\text{pri}, g, \text{act}} s' \], where \( s \) is the source state, \( s' \) is the target state, \( \text{pri} \in \mathbb{N} \) is the priority of the transition, \( g \) is the transition guard (predicate), and \( \text{act} \) is the transition action. A state transition is engaged when its source state is active and its guard evaluates to the Boolean value true, whereupon the action is executed, which in turn results in the target state. If a state has several outgoing transitions, the transition guards are evaluated according to the transition priorities. Transition sets which are lacking priorities are evaluated in a random order.

The execution semantics of AADL models is essentially defined in terms of scheduling and execution of threads. An active thread is initially in an “awaiting dispatch” state. Dispatches of the thread are triggered according to the specified dispatch conditions. Dispatches of periodic threads are solely triggered by a clock according to the time interval (period) specified with the thread. Dispatches of aperiodic and sporadic threads are triggered by the arrival of an event. In either case, an input-compute-output model of execution is triggered. Input data on in ports is by default locked at the time of dispatch, where new arrivals of data are inaccessible for the remainder of the current dispatch. Output on out ports, on the other hand, is transmitted through the connections at the time of thread completion, deadline, or at specific output times according to an Output_Time property.

2.1.1 Sensor-To-Actuator AADL Example

A simplistic example of a sensor-to-actuator system modeled in AADL is presented in Tables 2.2–2.4. A graphical representation of the model is presented in Fig. 2.1 to facilitate comprehension.
Figure 2.1: Graphical representation of the sensors-to-actuator AADL model.
Table 2.2 presents a system composed of a software process and a hardware platform that is connected to two sensors, an actuator, and a display through a CAN (Controller Area Network) bus. The `Actual_Processor_Binding` property specifies that the process is allocated to the processor.

Table 2.2: AADL sensor-to-actuator system component.

```aadl
system SensorToActuator
end SensorToActuator;

system implementation SensorToActuator.Impl
subcomponents
software: process ApplicationSoftware.Impl;
processorPlatform: processor ProcessorPlatform.Impl;
primarySensor: device primary_sensor;
secondarySensor: device secondary_sensor;
electric_actuator: device actuator;
digital_display: device display;
CAN_bus: bus CAN;
connections
CAN_Processor: bus access processorPlatform.external_bus -> CAN_bus;
CAN_Sensor1: bus access primarySensor.external_bus -> CAN_bus;
CAN_Sensor2: bus access secondarySensor.external_bus -> CAN_bus;
CAN_Actuator: bus access electric_actuator.external_bus -> CAN_bus;
CAN_Display: bus access digital_display.external_bus -> CAN_bus;
properties
Actual_Processor_Binding =⇒ (reference (processorPlatform)) applies to software;
end SensorToActuator.Impl;
```

The description of the software process is presented in Table 2.3. The process is composed of three threads, `Sensor`, `Logics`, and `Actuator`, that are connected by five port connections, `sensor_1`, `sensor_2`, `control`, `setpoint`, and `warning`. Each port connection is declared as immediate, i.e., the sending thread writes data to the connection at the time of its execution completion. Connections `sensor_1`, `sensor_2`, and `control` are also associated with a latency property that specifies the expected minimum and maximum duration from when data is written by the sender to when it is read by the receiver. Moreover, the process contains a data component, `Sensor_Failure`, that is shared between `Logics` and `Actuator`.

`Sensor`, as specified by its interfaces, properties, and behavioral specification, periodically outputs two integers that represent outputs from the two sensor devices. For simplicity, each sensor may either output the value “1” or the value “3”. These values are transmitted to `Logics` through connections
sensor 1 and sensor 2. Logics, as specified by its interfaces, properties, and behavioral specification, periodically uses these outputs to compute a control signal for positioning of the actuator device, where the mean of the two sensor values is used if they differ whereas the value from the primary sensor is used if they are considered as equal. If the sensor values differ, the shared data component Sensor_Failure is set. Actuator finally acts as an interface to the actuator and display, which periodically reads incoming requests and controls the actuator and display accordingly with one exception. If the received control signal is below or equal to the value of three, Actuator positions the device accordingly and displays the state of Sensor_Failure. If the signal is higher, no actions are performed.

Table 2.3: AADL sensor-to-actuator software components.

```
process ApplicationSoftware
features
  Actuator_value: out data port Base_Types::Integer;
  Display_warning: out data port Base_Types::Boolean;
end ApplicationSoftware;

process implementation ApplicationSoftware.Impl
subcomponents
  sensor: thread Sensor.Impl;
  logics: thread Logics.Impl;
  actuator: thread Actuator.Impl;
  sensorFailure: data Sensor_Failure;
connections
  sensor_1: port sensor.Sensor1 ->logics.Input1 {Timing =>Immediate; Latency =>0ms .. 4ms;};
  sensor_2: port sensor.Sensor2 ->logics.Input2 {Timing =>Immediate; Latency =>0ms .. 4ms;};
  control: port logics.Output ->actuator.Control_value {Timing =>Immediate; Latency =>1ms .. 6ms;};
  setpoint: port actuator.Actuator_setpoint ->Actuator_value {Timing =>Immediate;};
  warning: port actuator.Sensor_warning ->Display_warning {Timing =>Immediate;};
  logics_access: data access Logics.Sensor_Failure ->sensorFailure;
  actuator_access: data access actuator.Sensor_Failure ->sensorFailure;
end ApplicationSoftware.Impl;

thread Sensor
features
  Sensor1: out data port Base_Types::Integer;
  Sensor2: out data port Base_Types::Integer;
properties
  Dispatch_Protocol =>Periodic; Period =>14 ms; Priority =>2;
  Compute_Execution_Time =>1 ms .. 2 ms; Compute_Deadline =>14 ms;
end Sensor;
```
2.1 The Architecture Analysis and Design Language

thread implementation Sensor.Impl
annex behavior specification
{**
states
s0 : initial state;
s1 : state;
s2 : final state;
transitions
s0 \rightarrow s1 \{Sensor1:=1\}; s0 \rightarrow s1 \{Sensor1:=3\};
s1 \rightarrow s2 \{Sensor2:=1\}; s1 \rightarrow s2 \{Sensor2:=3\};
**}
end Sensor.Impl;

thread Logics
features
Input1: in data port Base::Types::Integer;
Input2: in data port Base::Types::Integer;
Output: out data port Base::Types::Integer;
Sensor_Failure: requires data access Sensor_Failure\{Access_Right => read, write\};

properties
Dispatch_Protocol => Periodic; Period => 16 ms; Priority => 1;
Compute_Execution_Time => 1 ms .. 2 ms; Compute_Deadline => 16 ms;
end Logics;

thread implementation Logics.Impl
annex behavior specification
{**
states
s0 : initial state;
s1 : final state;
transitions
s0 \rightarrow s1 \{Input1==Input2\} \rightarrow s1 \{Output:=Input1\};
s0 \rightarrow s1 \{Input1\Rightarrow=Input2\} \rightarrow s1 \{Output:=(Input1+Input2)/2;
Sensor_Failure=true\};
**}
end Logics.Impl;

thread Actuator
features
Control_value: in data port Base::Types::Integer;
Actuator_setpoint: out data port Base::Types::Integer;
Sensor_warning: out data port Base::Types::Boolean;
Sensor_Failure: requires data access Sensor_Failure\{Access_Right => read, write\};

properties
Dispatch_Protocol => Periodic; Period => 18 ms; Priority => 0;
Compute_Execution_Time => 1 ms .. 2 ms; Compute_Deadline => 18 ms;
end Actuator;

thread implementation Actuator.Impl
The description of the different hardware components, i.e., the processor platform, CAN bus, sensors, actuator, and display, is finally presented in Table 2.4. The processing platform has embedded RAM (Random Access Memory) and a fixed-priority preemptive scheduler. The processor, sensors, actuator, and display require bus access to the CAN bus.

Table 2.4: AADL sensor-to-actuator hardware components.
2.2 Control and Data Flow Analysis

A traditional approach of detecting erroneous behaviors of software systems is to analyze their control and data flows [35]. The control flow of a software system corresponds to the order in which the software instructions are executed. The data flow corresponds to the order in which data variables are assigned with values by instructions and how these variables and values subsequently are used. Control flow analysis is therefore a prerequisite for data flow analysis, since the flow of data is dependent on the order in which the instructions are executed. These flows are essential to the behavior of the system as they determine how system input is transformed into system output.

An AADL model expresses control and data flows throughout the software architecture in terms of component connections, subprogram calls, data accesses, and behavior specifications. For example, behavior specifications are composed of state transitions, each of which corresponds to an execution order of instructions: the guard of the transition is first computed and, if evaluated to true, the action (or sequence of actions) is executed. The control flow is typically represented by a directed graph in the analysis [36], where the vertices represent the instructions and the directed arcs between vertices represent the order of execution. Guards and actions are the executable instructions of behavior specifications and consequently yield the vertices of the control flow graph. An illustration of the control flow graph of a behavior specification is presented in Fig. 2.2. The “ENTRY” vertex represents the point through which
Chapter 2. Background Information

control enters. Given that $pri_2$ is higher than $pri_1$, $g_1$ is first evaluated ($s_1$ is the initial state), and if evaluated to $true$ ($T$), $act_1$ is executed. An execution of $act_1$ results in state $s_2$, where the transition from $s_2$ with the highest priority is executed in a similar sequence. On the other hand, if $g_1$ is evaluated to $false$ ($F$), the next transition from $s_1$ with the highest priority is executed in the fixed order. In this case, the transition with priority $pri_2$. The fixed execution sequence of transitions is repeated throughout the behavior specification until a final state is reached. An application of the analysis to the behavior specifications of the sensor-to-actuator model (see Fig. 2.1) is presented in Fig. 2.3.

```
annex behavior_specification
{**
states
s1 : initial state;
s2 : state;
s3 : state;
s4 : state;
...
sx : final ...
transitions
s1 [pri1] -[g1]-> s2 {act1};
s1 [pri2] -[g2]-> s3 {act2};
s2 [pri3] -[g3]-> s4 {act3};
... **};
```

Figure 2.2: Control flow illustration of a behavior specification.

The flow of data is analyzed by computing the definition-use pairs of the control flow graph. Assume $V_{def}$ is the set of vertices of the control flow graph that defines variable $var_i$, and $V_{use}$ is the set of vertices that uses $var_i$. For each pair of vertices $(v_x, v_y) \in V_{def} \times V_{use}$ such that there exists a control path $P = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ from $v_x$ to $v_y$ (where $v_1 = v_x$ and $v_n = v_y$) and any other vertex $v_z$ in $P$ does not define $var_i$, i.e., $v_z \notin V_{def}$ for $z = 2, 3, \ldots, n - 1$, there exist a data flow from $v_x$ to $v_y$. If the principle is applied to each variable of the control flow graph, all possible data flows are generated. As an example, the data flows of the behavior specifications of the sensor-to-actuator model is presented in Fig. 2.4. Note that these graphs provide an incomplete representation of the sensor-to-actuator architecture since the port connections and the shared data component are not included.
2.2 Control and Data Flow Analysis

Figure 2.3: Control flow graphs of the behavior specifications in Fig. 2.1.

Figure 2.4: Data flow graphs of the behavior specifications in Fig. 2.1.
2.3 Model Checking

Model checking refers to the verification of a finite-state concurrent system modeled in a precise mathematical language against a requirement specification of logic formulae [37]. A so-called model-checker, which exhaustively explores the model to determine whether it satisfies the logic formulae, is typically used to perform the verification. The UPPAAL model-checker [38], developed upon the theory of timed automata, is used in this thesis to explore and verify AADL models. UPPAAL extends the automata theory with the ability to model and verify timing properties, which is essential in the analysis of performance-critical embedded systems.

2.3.1 Modeling

A system is modeled in the UPPAAL environment as a network of timed (finite state) automata. A network of timed automata $NTA = \langle T\mathcal{A}, Var_G, Ch \rangle$ has a vector of $n$ timed automata $T\mathcal{A} = \langle TA_0, TA_1, \ldots, TA_{n-1} \rangle$, a set of shared (global) variables $Var_G$, and a set of synchronization channels $Ch$. A timed automaton $TA = \langle L, \ell_0, Cl, Var, I, E \rangle$ has a set of locations $L$, an initial location $\ell_0 \in L$, a set of real-valued variables $Cl$ called clocks, a set of local variables $Var$, a function assigning invariants to locations $I : L \to G$, and a set of instantaneous edges $E \subseteq L \times G \times Act \times U \times L$ of the form $\ell \xrightarrow{g,a,u} \ell'$. $G$ is a set of guards, which are Boolean expressions, possibly with logical connectives such as conjunction, disjunction, and negation, over variables and clock constraints of the form $x \text{expr} y$, where $x \in Cl \cup Var \cup Var_G$, $y \in \mathbb{N}$, and $\text{expr} \in \{<, \leq, =, \geq, >\}$. The Boolean evaluation of a guard determines whether an edge is enabled for execution or not (true=enabled, false=disabled).

A guard evaluation can only be conducted if the source state of the guarded edge is active. $Act = I \cup \delta \cup \{\tau\}$ is a set of input (denoted a?) and output (denoted a!) synchronization actions and the non-synchronization $\tau$, where an edge $\ell_0 \xrightarrow{g_0,a?u_0} \ell_0'$ of $TA_2$ is simultaneously executed in response to an execution of another edge $\ell_1 \xrightarrow{g_1,a!u_1} \ell_1'$ of $TA_1$, given that $\ell_0$ is active and $g_0$ evaluates to true at the time of execution of $\ell_1 \xrightarrow{g_1,a!u_1} \ell_1'$. $U$ is a set of updates, which are sequences of variable assignments of the form $v := \text{expr}$ and/or clock resets of the form $x := 0$, where $v \in Var \cup Var_G$, $x \in Cl$, and $\text{expr}$ is an arithmetic expression over integers.

In addition, locations may be labelled as urgent or committed. 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.

The semantics of a timed automata network is defined in terms of a timed transition system over system states. A system state is a triple \((\vec{\ell}, \overline{\phi}, \sigma)\) where \(\vec{\ell}\) is a location vector over all automata such that \(\vec{\phi}^0, \vec{\phi}^1, \ldots, \vec{\phi}^{n-1}\) denotes the current location of \(TA_0, TA_1, \ldots, TA_{n-1}\). \(\overline{\phi}\) is a clock valuation vector over all automata such that \(\overline{\phi}^0, \overline{\phi}^1, \ldots, \overline{\phi}^{n-1}\in\mathbb{R}_+^{n}\) and satisfies the invariants of the locations \((\overline{\phi} \models I(\vec{\ell}))\), and \(\sigma\) is a variable valuation vector that maps variables to values and \(\sigma \models I(\vec{\ell})\). The initial system state is a state \((\vec{\ell}_0, \overline{\phi}_0, \sigma_0)\) where \(\vec{\ell}_0\) is the initial location vector, \(\overline{\phi}_0\) maps each clock to zero, and \(\sigma_0\) maps each variable to its default value. Progress is made through delay transitions or discrete transitions. A delay transition is of the form \((\vec{\ell}, \overline{\phi}, \sigma) \xrightarrow{d} (\vec{\ell}, \overline{\phi} \oplus d, \sigma)\) where \(\overline{\phi} \oplus d\) is the result of synchronously adding the delay \(d\) to each clock valuation in \(\overline{\phi}\). Let \(\vec{\ell}[\ell'_i/\ell_i]\) denote that the \(i\)th vector element \(\ell_i\) is replaced by \(\ell'_i\). A discrete transition is of the form \((\vec{\ell}, \overline{\phi}, \sigma) \xrightarrow{u} (\vec{\ell}[\ell'_i/\ell_i], \ell'_j/\ell_j, \ell'_k/\ell_k, \ldots, \overline{\phi}', \sigma')\) such that there are edges \(\ell_{i/j/k} \ldots g_{i/j/k} \ldots a_{i/j/k} \ldots u_{i/j/k} \rightarrow \ell'_{i/j/k} \ldots\) where \(\overline{\phi}\) and \(\sigma\) satisfies \(g_1 \land g_2 \land g_3 \ldots\), the result of updating \(\overline{\phi}\) and \(\sigma\) according to \(u_1, u_2, u_3, \ldots\) is \(\overline{\phi}'\) and \(\sigma'\), and the edges are synchronous over complementary actions (\(a?\) complements \(a!\)). A trace is a sequence of states such that there exist a delay or discrete transition from each state in the sequence leading to its successor state.

The language has many similarities to transition systems of AADL behavior specifications, however where states and transitions in behavior specifications are referred to as locations and edges in timed automata. An example of mapping the AADL behavior specifications of Sensor, Logics, and Actuator (in Fig. 2.1) to UPPAAL timed automata is presented in Fig. 2.5. Besides mapping the states and state transitions of the behavior specifications, the dispatch protocols, interfaces, connections, and input/output behaviors are represented by clock constraints, variables, and variable assignments. The Awaiting dispatch locations are the initial locations (denoted by inner circles) of the automata and labeled with clock invariants (e.g. \(cl<=14\)), meaning that the automata may exist within the locations as long as the clock is less or equal to the quantities of time units. From each initial location, there is an edge to a location that represents the initial state of the corresponding behavior specification. The edges are guarded by clock constraints that overlap with the clock invariants such that they coincide with the periodicities of the threads. Firings of these edges are consequently forced when the clock reaches the thread periods. Furthermore, the edges are labeled with input assignments (except for Sensor) that represent
readings of input at the time of dispatch. Similarly, the locations that represent the final states of the behavior specifications are associated with edges that represent the output assignments. The locations are labeled as committed in order to force a transmission of output at the time of completion.

Figure 2.5: Timed automata examples of Sensor, Logics, and Actuator in Fig. 2.1.

2.3.2 Requirements Specification and Verification

A timed automata model is verified by UPPAAL through the specification of requirements in terms of Timed Computational Tree Logic (TCTL) formulae [39]. The UPPAAL model checker subsequently searches the state space...
of the model to check whether it satisfies the formulae (requirements). UP-PAAL is able to model-check three types of formulae:

- Reachability formula: checks whether a predicate eventually is satisfied by a reachable state along some path.
- Safety formula: checks whether a predicate invariantly is satisfied in each state of some path or all paths.
- Liveness formula: checks whether a predicate eventually is satisfied by a reachable state in all paths.

A predicate is a Boolean expression such as \( \text{setpoint} = 2 \) and \( cl > 16 \), where these formulae are satisfied whenever \( \text{setpoint} \) equals two and \( cl \) is larger than sixteen time units. Reachability properties are verified by temporal operators \( E \) (pronounced “for some path” or “exists one path”) and \( <> \) (pronounced “eventually”). For example, in order to verify that the predicate \( \text{warning} = \text{true} \) is reachable, i.e., that the system can display a warning, the formula \( E <> \text{warning} = \text{true} \) (pronounced “for some path eventually \( \text{warning} = \text{true} \) holds”) may be used.

Safety properties are verified by temporal operators \( A \) (pronounced “for all paths”), \( E \), and \( [] \) (pronounced “always” or “globally”). Formula \( A[]p \) (pronounced “for all paths globally \( p \) holds”), where \( p \) is a predicate, 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. For example, in order to verify that the position of the actuator always is set to a value that is less than or equal to the value of three, the formula \( A[]\text{setpoint} \leq 3 \) may be used.

Liveness properties are verified by 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. For example, an eventual dispatch of \( \text{Actuator} \) at 18 time units for all paths can be verified by \( A <> (\text{Actuator.s0} \text{and } cl = 18) \), and an eventual warning signal whenever the output of the two sensors differ may be verified by \( (\text{Sensor.s2} \text{and } \text{Sensor.Sensor1} = \text{Sensor.Sensor2}) \rightarrow \text{warning} = \text{true} \).

The result of verifying these properties in the UPPAAL environment is presented in Fig. 2.6. Each formula is satisfied by the model except for the formula that checks whether differences between the two sensor values always lead to a
Figure 2.6: Model checking results.

displayed warning \((\text{Sensor.s2 and Sensor.Sensor1} \neq \text{Sensor.Sensor2}) \rightarrow \text{warning} == \text{true})\). The problem is that the model does not include mechanisms that regard scheduling, priorities, execution time, concurrency, and context switches of threads, as specified by the AADL model. Furthermore, the shared data component, “Sensor_Failure”, is specified with a concurrency protocol that controls accesses to the data through a semaphore. Consequently, the sequence of thread executions, accesses to shared data, and thread completions may occur at any time after dispatches, as illustrated in the execution path presented in Fig. 2.7. Sensor, which has the shortest period and the highest priority, executes according to the properties. Logics subsequently dispatches but does not complete its execution before the dispatch of Actuator, which, despite being specified with a lower priority in the AADL model, executes and completes prior to Logics. Data is therefore not transmitted as intended between Logics and Actuator, which causes the omission of a warning. The source of the fault is consequently not an incorrect AADL model, but an incorrect representation of it in timed automata. A correct representation is therefore a critical prerequisite for model checking of AADL models, where properties such as scheduling, concurrency, context switches, execution time, and protocols of shared resources, are included in the transformation.

Another issue of the presented model checking example is the lack of behavior coverage assurance. For example, according to the control flow graphs presented in Fig. 2.3, Sensor is designed with four different execution paths and Logics and Actuator are designed with two each. Although the presented formulae verifies different properties of the architecture, the extent to which the execution paths have been covered in the verification is inconclusive. In fact, the input of Actuator (Control_value) can never be higher than three, which implies that transition \(s_0 \xrightarrow{\text{Control_value} > 3} s_1\) cannot be fired, indicating a possibly incomplete or superfluous design. Moreover, the data port connections between the threads are declared with minimum and maximum latencies, where the required minimum latency of control in the worst case scenario is
violated by the design. The problem is that *Logics* has a worst case execution time of two milliseconds and that *Actuator* is dispatched two milliseconds after the dispatch of *Logics*. The latency is therefore in worst case less than the required minimum. There are consequently faults within the design that cannot be detected by the requirements specification presented in Fig. 2.6 due to an incomplete coverage of its possible behaviors. A requirements specification that enforces a complete coverage of the architectural design, in addition to a correct representation of it, is therefore essential to detect all possible faults.

Figure 2.7: Example of incorrect scheduling of the sensor-to-actuator model.
2.4 Model-based Testing

Testing of software and hardware systems has traditionally been carried out manually or in a semi-automated manner. Manual testing is usually conducted by reading requirements and system specifications in order to directly experiment with the system under test (SUT), to test whether it behaves according to the expectations or not. Semi-automated testing is usually conducted by reading requirements and system specifications in order to design test cases that are encoded into test scripts. Test scripts can then be automatically executed against the SUT. A test case is essentially composed of a set of system inputs and, given the inputs, a set of results (outputs) the system is expected to produce. Even though test scripts may be automatically executed against the SUT for initial or regression testing, the disadvantages of manual test case design, i.e. the risk of human error (e.g. missed tests, incorrect tests, redundant tests, and incomplete coverage) and the cost of human labor, are crucial to overcome for large and complex systems.

A growing research effort has been carried out in the field of model-based testing (MBT) to increase automation and rigorousness of testing. Efforts enclose both the 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 [40]. MBT constitutes testing of software based on computer-readable models that represent the expected behavior of the SUT, typically comprising an input domain, an expected output range, and mappings between the two. Each mapping may serve as a test case, which can be automatically generated and, if the testing harness is connected to the SUT, automatically executed. In this thesis, the research of architectural model-based testing is focused on the automatic generation of test cases, but not the automatic execution of them. The latter challenge require case-specific details in order to develop a solution, as the interface between an embedded system and its environment may differ greatly between products, enterprises, and industries.

Models for MBT are commonly represented by languages based on automata theory [41], where test cases are generated according to some coverage criterion, such as state coverage, transition coverage, predicate coverage (covering all possible truth and false values of branching expressions), and path coverage. State space searches of the model are subsequently performed to accumulate traces that cover the model according to the coverage criterion, where the search may be performed by a model checker [28]. The traces may subsequently be used to generate tests as they include information about the
expected results with respect to the different inputs. For example, consider
the execution path of the sensor-to-actuator system illustrated in Fig. 2.8. As-
sume the threads are executed in compliance with the scheduling properties,
i.e., Sensor \rightarrow Logics \rightarrow Actuator. The corresponding trace from a search of
the end-to-end path is illustrated in Fig. 2.9. Given that the model represents
the required behavior, the trace specifies that an implementation of the design
should display a warning and set the actuator to “2” when it is stimulated with
an input of “1” and “3”. In order to completely test that a system meets the
required behavior, traces from all possible paths through the model must be
extracted and tested against the SUT.

![Diagram of execution path](image)

(a) Sensor

(b) Logics

(c) Actuator

Figure 2.8: Illustration of an execution path of the sensor-to-actuator system.
Figure 2.9: Illustration of the trace generated from the execution path presented in Fig 2.8.
2.5 Regression Verification

A development process will likely include modifications of artifacts that already have undergone verification, 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. The term regression verification is used in this thesis as a collective term for this type of verification, including the thoroughly studied area of regression testing [42]. 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 [43]. Since a modification may not affect every functional and non-functional property of the system, it may not be necessary to re-execute every previous test case. A number of approaches has been developed in the areas of regression test case selection, test case prioritization, and test suite minimization, to effectively and efficiently perform regression testing [42]. Regression test case selection approaches are used to identify tests that are not relevant with respect to a modification. Regression test case prioritization approaches are used to identify tests that are most likely to detect faults that may have been inserted by the modification. Test suite minimization approaches seek to optimize a regression test suite by removing redundant test cases.

2.6 Program Slicing

The notion of determining tests that are unnecessary to re-execute in response to a modification is referred to as selective regression testing [43]. This is an important challenge to industry as regression testing frequently is performed [44], where test suites may contain large numbers of test cases. The method of program slicing may be used to select tests that are necessary to re-execute [45]. The goal of slicing, originally defined by Weiser [46], is to remove instructions and variables that do not have an effect on and are not affected by the value of a variable at some program point (statement or instruction) referred to as the slicing criterion. By using the modification of a software system as the slicing criterion, the parts of the system that are not impacted by the modification can be traced such that verification executions that do not cover the modification or impacted parts are disregarded in the regression verification process. Causal relationships to the value of a variable at some point are determined by the control and data dependencies between the instructions of the software system.
These are commonly derived from control and data flow analyses [47], where a vertex (instruction) $v_2$ of a control flow graph is control dependent on a preceding vertex $v_1$ if the execution of $v_1$ determines whether $v_2$ shall be executed or not. For example, consider the control flow graph of Actuator in Fig. 2.10. The outcome of $\text{Control\_value} \leq 3$ determines whether $\text{Control\_value} > 3$ or $\text{Actuator\_setpoint} := \text{Control\_value}$, $\text{Sensor\_warning} := \text{Sensor\_Failure}$, and $\text{Sensor\_Failure} := \text{false}$ should be executed. These instructions are consequently control dependent on $\text{Control\_value} \leq 3$. Given that $\text{Control\_value} \leq 3$ is modified, both execution paths have to be considered in the regression verification process. However, the execution of $\text{Actuator\_setpoint} := \text{Control\_value}$, $\text{Sensor\_warning} := \text{Sensor\_Failure}$, and $\text{Sensor\_Failure} := \text{false}$ is independent to $\text{Control\_value} > 3$ (and vice versa), where a modification of $\text{Control\_value} > 3$ does not require any reverification of $\text{Actuator\_setpoint} := \text{Control\_value}$, $\text{Sensor\_warning} := \text{Sensor\_Failure}$, and $\text{Sensor\_Failure} := \text{false}$. Data dependencies are synonymous to definition-use pairs, where a vertex $v_2$ is data dependent on a preceding vertex $v_1$ if $v_2$ uses a variable defined by $v_1$, i.e., the result of $v_2$ depends on the result of $v_1$. According to the data flow graph of Actuator in Fig. 2.10, both execution paths are data dependent on the input port. Any modification of the input interface, or any modification that impacts the input interface, consequently requires reverification of both execution paths. The output ports, on
the other hand, are data dependent on instructions pertaining to only one execution path: \( \text{Actuator.setpoint} := \text{Control.value} \) and \( \text{Sensor.warning} := \text{Sensor.Failure} \). Modifications of these instructions require reverification of all other instructions that that are dependent on the output interfaces.

The control and data dependencies are commonly represented in directed graph referred to as a dependence graph [47]. The parts that have causal relationships to the value of a variable are then determined by computing the transitive closure of the dependence graph with respect to the slicing criterion (the modified vertex). The parts that do not have an effect on the value of the variable are removed by computing the backward transitive closure, known as backward slicing. The backward transitive closure is determined by all elements that are backward reachable (through the arrows of the graph) from the slicing criterion. Elements of the graph that are not backward reachable cannot impact the modification (slicing criterion) in terms of control and data. The parts that are not affected by the slicing criterion are removed by computing the forward transitive closure, known as forward slicing [47]. Elements that are not forward reachable cannot be impacted by the modification in terms of control and data.
Chapter 3

Research Methodology

3.1 Research Method

The scientific method applied in this research project is composed of six consecutive and iterative stages, as illustrated in Fig. 3.1. The preliminary research is composed of literature reviews, industrial questionnaires, and interviews with domain experts and used as basis for the derivation of research challenges and goals. Innovative methods are subsequently developed with the aim of achieving the research goals. These methods are subsequently subjected to initial tests in pilot experiments with artificial systems. Depending on the compliance of the results with respect to the expectations, the methods are either accepted, improved, or rejected and replaced. Replacements and improvements are contrasted with additional small-scale experiments until satisfactory results are obtained.

Methods which yield satisfactory results are subsequently implemented within a computer tool, to enable an easy adoption into industrial practices. Evaluation of the tool is finally conducted through empirical large-scale and industrial studies.
Figure 3.1: Flowchart of the research methodology
3.2 Research Challenges and Goals

The overarching challenge of this research project is to increase quality of critical embedded systems and reduce the cost of the development processes. As motivated by studies within this area, e.g. [15][18][19][20][21][24][25][5], there is a need for a holistic, systematic, reliable, and automated verification process that effectively and efficiently detects architectural engineering faults in the development of complex and critical embedded systems. In order to reach such a solution, the research project focuses on extending and integrating both theoretical and practical knowledge within the areas of dependability, model-driven engineering, architecture description languages, formal methods, automated verification, and regression verification. In this section, the general challenge is decomposed to its elements with formulations that can be scientifically studied. Furthermore, a corresponding research goal is defined for each elementary challenge.

**Challenge 1:** The first challenge is related to the fundamental problems of verification: the objective, the data selection, the coverage criteria, and the oracle [48]. Firstly, it is crucial to thoroughly identify why the analysis should be performed (the objective). The common denominator of architecture-based verification is to assure correctness, consistency, and completeness of system architectures [49, 50]. This objective must be decomposed to atomic properties to be compatible with formal methods. Secondly, the types of data that are interesting with respect to the objective and how they are localized in the architectural design must be identified (the data selection). Thirdly, the relative quantity of samples to analyze must be identified to know when to stop the analysis (the coverage criteria). And finally, the expected result of the analysis must be identified to be able to verdict if a result is acceptable or not (the oracle). These verification criteria must be addressed unambiguously to be suitable for formal and automated verification.

- **Goal 1:** Develop unambiguous verification criteria for the verification of architectural correctness, consistency, and completeness.

**Challenge 2:** The second challenge is that formal architecture-based verification requires architectural models with formally defined semantics. The AADL standard provides a detailed description of the semantics, but it is not formally defined. Consequently, the semantics of AADL has to be formalized in order to enable applications of formal methods. Furthermore, the AADL execution model consists of numerous different aspects of a run-time system, such as synchronous and asynchronous interactions, recovery execution, var-
ious scheduling policies and resource sharing protocols, preemptive and non-preemptive scheduling, immediate and delayed connections, etc. A formalization of each possible execution model configuration is out of scope of the thesis work, where a subset suitable for the targeted type of systems must be selected. The challenge of formalizing AADL semantics also includes the problem of selecting a formal domain that is suitable for representing AADL semantics and for computer-aided verification.

- **Goal 2:** Select a subset of the AADL execution model that is suitable for critical embedded system and a formal domain that is suitable for AADL semantics and computer-aided verification. Subsequently define the mapping from the AADL subset to the formal domain.

**Challenge 3:** The third challenge is to develop techniques that implement the verification criteria in the selected formal domain such that they can be automatically executed by a computer tool. Firstly, the data of the architectural design that should be subjected to verification must be localized in the formal domain. Secondly, ensuring coverage of data samples in model checking and model-based testing processes require an instrumentation of the formal model that monitors the analysis of it. Finally, state space searches of the formal model for the detection of faults or the generation of test cases must be driven by some propositional formula, according to the oracle.

- **Goal 3:** Develop model checking and model-based testing techniques that implement the verification criteria in the formal domain.

**Challenge 4:** The fourth challenge is to perform regression verification efficiently, where architectural parts that are not impacted by the modification are not unnecessarily re-verified. The challenge requires identification and impact analysis of architectural design changes such that verification runs that do not cover impacted parts can be deselected in the regression verification process. The assessment of whether a verification run cover a modification or not, on the other hand, requires traceability between the previously executed verification runs and their architectural coverage. The overhead expense of performing the selection process, in addition to re-execution of the selection, must not exceed the cost of a re-run all approach to be efficient.

- **Goal 4:** Develop a regression verification technique that selectively re-executes only those verification runs that may be impacted by the modification.
3.2 Research Challenges and Goals

Challenge 5: The fifth challenge is to implement the developed framework into a computer tool, to enable easy adoption into industrial practices and to evaluate whether the solution meets the expectations or not. Furthermore, the goal of this work is essentially to increase the detection rate of faults and to reduce the cost of verification. Thus, the fault-detection effectiveness and resource efficiency of the framework and tool must be evaluated. Furthermore, formal verification is commonly limited in their scalability to complex systems due to the state explosion problem [29]. Evaluations of both utility and scalability are therefore critical.

- **Goal 5:** Develop a computer tool that implements the developed verification framework and evaluate the utility and scalability of the solution.
Chapter 4

Related Work

4.1 Verification Criteria

Garlan and Shaw [51] recognized the emerging field of architectural engineering to cope with the increasing complexity of systems. A system architecture, as described by Garlan and Shaw, is the collection of components and the interactions among them. From an architectural perspective, the primary verification objectives are to ensure consistency (no contradictions), correctness (compliance with requirements), and completeness (no lack or excess) of component interfaces and the control and data interactions among them [49][50]. Achievement of extra-functional requirements, in addition to functional requirements, can in turn be ensured as long as each individual component complies with its interface specification.

One or more coverage criteria are commonly applied when software is analyzed, to measure and control the extent to which a program is exercised by the analysis [52]. Examples of some basic coverage criteria are: statement coverage, decision (branch) coverage, and condition (predicate) coverage. Statement coverage is the percentage of statements within the software that have been executed in the analysis. Decision coverage is the percentage of branches of the control flow. Condition coverage is the percentage of branching expressions which atomic conditions have been evaluated both to true and false in the execution. This list essentially represents an increasingly rigorous coverage of the system under analysis, where functional safety standards typically require more thorough coverage criteria the higher the risk associated with the system is. For example, at the software unit level, ISO 26262 [30] highly recommends...
statement coverage for units with a safety integrity level in the lower end of the spectrum and branch coverage for units in the higher end. Note that 100% branch coverage implies 100% statement coverage, as defined by ISO 26262. Other types of traditional coverage criteria are data-flow coverage [53], i.e. the percentage of executed definition-use pairs, and basis path coverage [54], i.e. the percentage of executed linearly independent paths, which guarantees complete branch coverage without having to cover all possible control flow paths. Since a software system may include loop constructs yielding an infinite number of potential execution paths, complete coverage is often not feasible even in theory. Restrictions to a limited path coverage might therefore be necessary.

At the architectural level, ISO 26262 requires control and data flow analysis of the component interactions, measured in terms of, for example, percentage of executed function calls. Jin and Offut [55] have proposed the idea of adapting traditional control- and data-flow coverage criteria to the architectural level, to ensure a thorough analysis of the potential component interactions. The concept is that all control- and data-flow paths through components and component connections, as prescribed by the specified system structure, must be exercised in the verification processes. In this thesis, this concept is utilized to developed architectural verification criteria for systems modeled in AADL.

A common approach of deriving control- and data-flow verification data is to represent the system behavior as a directed graph, referred to as a control flow graph, initially defined by Allen [36]. The vertices in the graph correspond to the executable expressions whereas edges correspond to the sequences in which the executable expressions might be traversed at runtime. The approach is adopted in this thesis to derive the necessary verification data from AADL models.

### 4.2 Formal Verification

Berthomieu et al. [56] presents ongoing work of transforming AADL models with fixed-priority threads into timed transition systems (TTS) through an intermediate Fiacre model, which in turn can be checked by the Tina toolbox. The authors report that the problem of addressing scheduling and time-constrained behaviors is not solved by their solution.

Murugesan et al. [57] propose an approach to compositional verification of AADL models against requirements expressed in past-time linear temporal logic (PLTL) by means of the AADL model-checker AGREE. The approach is compositional in the sense that component-level behavior is described in
Simulink and verified by the Simulink Design Verifier [58]. The verification techniques are limited with respect to extra-functional properties as concurrent execution, shared resources, scheduling, and timing properties are not considered.

A mapping of AADL to Petri Nets is presented by Renault et al. [59], where the objective is to verify that the system is free from deadlocks and that defined data interactions behave correctly. Based on the presented principles, the approach only supports event-driven, non-preemptive, architectures and does not include timing properties in the analysis.

Björnander et al. [60] present the tool ABV, which provides model checking of AADL models against computation tree logic (CTL) formulae through a transformation to ML (Meta Language). The work encompasses timing properties but the selected subset does not include execution semantics of processes and threads, where the architectural analysis is restricted to abstract system components without concurrent execution.

Chkouri et al. [61] present the tool “AADL to BIP”, which provides model checking of AADL models through a transformation to BIP (Behavior Interaction Priority). It is not evident to which degree concurrency of software processes is supported as information on how scheduling properties are captured in BIP is not presented.

Inverardi et al. [62] present CHARMY, a tool for UML-based modeling and analysis of software architectures. By means of a transformation to Promela code (Process Meta Language), the SPIN model-checker [63] is used to verify temporal properties of the architectural model.

Singhoff et al. [64] present Cheddar, which provides a schedulability and resource requirements analysis feature for AADL models. Esteve et al. [65] present COMPASS, which provides model checking of SLIM models, a variant of AADL, through a transformation to Markov chain.

### 4.3 Model-based Testing

No notable contribution of automatic test case generation from AADL models have been recognized prior to 2017. Deng et al. [66] propose an automatic test case generation approach based on branch coverage of AADL behavioral models. However, the approach does not consider semantics of the core language, only the semantics of the AADL behavioral annex, where conformance of functional properties in conjunction with extra-functional cannot be tested.

Test case generation techniques for other ADLs have been proposed in
Chapter 4. Related Work

[67][55]. The dynamic semantics of these ADLs are either not based on any formal theories, which is solved through model transformations. In [67], the model-based testing technique is based on Petri nets models which are transformed from Acme descriptions. In [55], the technique is based on Behavioral Graphs (BGs), an extension of Petri nets, transformed from Wright descriptions. Neither describe how code-level test cases with concrete test data are generated, but rather the architectural paths from which concrete test cases should be generated.

Muccini et al. [68] present a generic approach to conformance testing. Their approach is based on capturing the architectural behavior in a labeled transition system and subsequently deriving a set of paths with an appropriate coverage of the architecture. These paths are then considered as abstract test cases, which can be converted to code-level tests by manually annotating them with concrete data values. The principles of how the architectural behavior, in particular timing properties and concurrency, is captured in a transition system are not presented. It is consequently difficult to evaluate the applicability of the approach to an ADL and to what degree it can be automated.

4.4 Regression Verification

No related work within the area of regression verification based on AADL models have been published. Muccini et al. [69] explore how selective regression testing generally can be performed at the software architecture level to reduce the cost of verifying changes of architectures. The method used is based on the comparison of graphs representing a system and its modified version such that only test cases that cover changed nodes are selected for regression verification. The approach does not consider the impact a change may have on the remaining architecture and therefore must be complemented with change impact analysis to be reliable.

Harrold [25] suggests slicing at the architectural level as a means of identifying the parts that are affected by a change and must be covered in the regression verification process. Ottenstein and Ottenstein [70] showed how program slicing algorithms could be defined in terms of operations on so called program dependence graphs (PDGs) – a method also used for effective slicing of specifications and models [71]. A PDG is a directed graph of vertices and edges, where vertices represent the executable expressions of a monolithic program and edges represent control and data dependencies among those vertices. To be able to perform program slicing in the more general case, where a program...
consists of multiple procedures, Horwitz et al. [72] introduced the so called system dependence graph (SDG). SDGs extend the expressiveness of PDGs such that procedure calls and parameter passing by value can be integrated.

4.5 Verification Framework

The formal verification approaches presented in Section 4.2 can be used to automatically detect incorrect architectural designs by checking if they comply with requirements specifications expressed by logic formulae. Given that the architectural model-based testing techniques presented in Section 4.3 are improved and implemented, such that concrete test cases are generated, they can be used to automatically detect functional implementation faults. Finally, given that techniques are implemented according to the regression verification approaches presented in Section 4.4, they can be used to facilitate selective regression verification of architectural design changes. Nevertheless, the techniques and tools are developed upon different formalisms and semantic domains, making them impossible or difficult to combine in a development process without losing semantic consistency and traceability. Any loss of consistency or traceability hinders the effectiveness and efficiency of the joint verification process. Semantic inconsistency may result in incoherent expectations of the architectural implementation with respect to the architectural design, thereby decreasing the effectiveness of finding faults, and even creates potentially undetectable false positives (declarations of non-existing faults) and false negatives (non-declarations of existing faults). Traceability, on the other hand, is necessary to efficiently handle regression verification of architectural design changes, not least to be able to build sound and structured certification arguments. In addition, the related work excludes scheduling, protocols of shared resources, real-time constraints, or concurrency through multitasking and parallel processing in the analysis, which often are central to architectural engineering of critical embedded systems. In the contributions of this thesis, functionality is analyzed in conjunction with these properties, such that a solution to the limitations of the related work is provided. Furthermore, the formal verification techniques and tools presented in Section 4.2 do not include methods that measure and enforce coverage of the architectural design in the analysis, which is essential to determine the extent to which an architecture has been verified. The verification techniques presented in this thesis measure and enforce coverage criteria in terms of architectural control and data flow paths in the analysis.
Inverardi et al. [62], which present the tool CHARMY for UML-based modeling and verification of software architectures by means of a transformation to Promela code and the SPIN model-checker [63], are planning to provide a more comprehensive tool in their future work, where it is extended with dependence analysis and architectural slicing.

Simulink Design Verifier [58] provides a formal verification and analysis framework similar to the contribution of this thesis, however for Simulink models. Besides the ability to automatically detect design faults and requirements violations through model checking, the tool includes a condition, decision, and modified condition/decision test data generator and coverage analyzer and a slicer for dependency tracing and variability modeling.

4.6 Hazard Analysis

Grunske and Han [73] suggest that industry needs solutions for automated analyses of reliability, availability, and safety, such as of error propagations and probabilities of hazardous events, and automated generations of related work products, such as fault trees and Failure Mode and Effects Analysis (FMEA) tables, based on architectural designs. Ern et al. [74] propose an approach to formally and automatically generate FMEA tables from AADL models. The approach is compatible with AADL models wherein the AADL error model annex [75] is used to describe anomalies with respect to the nominal behavior. The error model annex essentially enables modeling of both erroneous and non-erroneous states of the system. The transition from a non-erroneous state to an erroneous state is specified by the erroneous event that triggers the transition, such as the arrival of a faulty input to a component. Moreover, the potential erroneous output in response to the arrival of an erroneous state may be specified. The approach presented by Ern et al. focuses on the potential errors that may occur according to the model, and identifies the erroneous transitions that are activated subsequent to their occurrences. The identified chains of events and transitions are subsequently annotated to the corresponding entries in the FMEA table. Björnander et al. [76] introduces a method that uses predicate logic to formalize safety goals of safety cases [7] expressed by the Goal Structuring Notation (GSN), and verifies these against the corresponding architectural designs expressed by AADL. Note that the work does not support hazard analysis explicitly, but rather the verification of whether safety goals derived from hazard analyses are satisfied by the architectural design.

The contribution of this thesis includes a formal and automated impact
analysis technique such that model-checking and model-based testing processes may be executed selectively in response to design changes. The technique determines the impact of a change based on an analysis of interdependencies between the system components. Although challenges of hazard analysis are not explicitly targeted in the problem formulation of this thesis, the dependence analysis provided by this thesis may be utilized for automated and formal hazard analysis, such as the identification of potential error propagations, single point failures, and common cause failures.

It should be noted that there exists related tools, techniques, and studies that are not available for review through publications, and consequently not presented as related work within this chapter.
Chapter 5

Summary of Contributions

The main contribution of this thesis is the Architecture Quality Assurance Framework (AQAF) illustrated in Fig. 5.1 and a corresponding tool support, the Architecture Quality Assurance Tool (AQAT). AQAF is a theory developed to provide a holistic, systematic, formal, and automated verification process for architectural engineering of critical embedded systems, from requirements analysis and design to implementation and maintenance. The framework is composed of an architectural model checking technique to detect design faults, an architectural model-based test case generation technique to detect implementation faults, and an architectural selective regression verification technique to efficiently detect faults introduced by maintenance modifications. AQAT is an application program that implements the theory of AQAF and provides an easy adoption into industrial practices. In this chapter, overviews of the different framework components, the tool, and their contributions to the research goals specified in section 3.2 are presented in relation to the list of papers included in the thesis. Overviews of contributions in terms of evaluation are presented separately in Chapter 6.

Table 5.1 presents an overview of the different contributions to the research goals and the corresponding papers in which the work is presented.
Figure 5.1: Flowchart of the verification framework. A black shape denotes a necessary framework input. A gray shape denotes a formally defined process or rule set. A white shape denotes an artifact produced by the framework.

<table>
<thead>
<tr>
<th>Research goal</th>
<th>Contribution (section)</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal 1</td>
<td>Architecture flow graphs (5.1)</td>
<td>D</td>
</tr>
<tr>
<td>Goal 1</td>
<td>Verification criteria (5.2)</td>
<td>A</td>
</tr>
<tr>
<td>Goal 2</td>
<td>Formal semantics (5.3)</td>
<td>B</td>
</tr>
<tr>
<td>Goal 3</td>
<td>Model checking with observer automata (5.4)</td>
<td>D</td>
</tr>
<tr>
<td>Goal 3</td>
<td>Model-based testing with observer automata (5.5)</td>
<td>D</td>
</tr>
<tr>
<td>Goal 4</td>
<td>Selective regression verification (5.6)</td>
<td>C</td>
</tr>
<tr>
<td>Goal 5</td>
<td>The architecture quality assurance tool (5.7)</td>
<td>E</td>
</tr>
<tr>
<td>Goal 5</td>
<td>Evaluation of the framework and tool (6)</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 5.1: Relations between goals, contributions, and paper A to F.
The primary verification objective is to ensure consistency, correctness, and completeness of component interfaces and the control and data interactions among them. In order to address the data selection problem of research Goal 1, AQAF includes a technique, presented in paper D, that captures all prescribed control and data flows of an architectural design (AADL model) in a directed graph referred to as the architecture flow graph (AFG). Architectural control flows refer to the orders in which software components and their instructions are executed. Architectural data flows refer to the orders in which data variables, including interfaces, connections, and shared data objects, are assigned by a component and subsequently used, possibly by a different component. All such flows, which constitute the prescribed behaviors of the architectural design, must be analyzed in the verification processes to achieve the objective. This is of industrial importance as contemporary functional safety standards, such as ISO 26262 [30], require control and data flow analysis of architectural designs. Moreover, this type of information is necessary for conducting rigorous impact analyses of design changes, through control and data dependencies. The contribution therefore also supports the accomplishment of Goal 4.

The AFG generation technique is an extension of system dependence graphs (SDGs) generation developed by Howritz [72], which comprises extraction of control and data flows within and between software procedures invoked by calls. Since the notion of multi-threaded software systems with data passing by both reference and value is not addressed by SDGs, the concept is extended with these elements such that they are applicable to software architectures of modern embedded systems.

The AFG of an AADL model is created through three steps. The first step is to create a control flow graph (CFG) for each thread and subprogram component of the design, as described in Section 2.2. This entails isolated analyses of the behavioral models of the components to compute their prescribed component-internal control flows, which constitute the arcs of the CFGs. The second step is to compute the prescribed component-internal data flows of each component as described in Section 2.2 and merge them to the corresponding CFG. The resultant graphs, including both component-internal control and data flows, are referred to as program flow graphs (PFGs). Similarly to step one, the analysis is performed on each component in isolation. The third and final step is to integrate the PFGs according to the component connections, resulting in the AFG of the AADL model.

Both data and control may be transmitted between AADL components.
Control is sent to threads through event ports and event data ports, whereas subprograms are invoked through calls. In either case, control is transferred to the entry point of the receiver. Data is sent through data ports, event data ports, subprogram parameters, and shared data components. The interactions are by default constrained to the input-compute-output semantics of AADL. Input data from connections are assigned to in ports and in parameters at the time of dispatch of the receiver. Output data on out ports and out parameters, on the other hand, is assigned to connections at the time of completion of the sender. Consequently, input assignments coincide with entry vertices whereas output assignments coincide with exit vertices. These inter-component flows are represented in an AFG by four distinguished vertex types similarly to system dependence graphs defined by Horwitz et al. [72]: (1) \textit{actual-in} vertices of the form \textit{connection} = \textit{out\_interface} to represent assignments that transfer data of output interfaces to connections; (2) \textit{formal-in} vertices of the form \textit{in\_interface} = \textit{connection} to represent assignments that transfer data of connections to input interfaces; (3) \textit{formal-out} vertices of the form \textit{connection} = \textit{out\_parameter} to represent assignments that transfer data of a callee’s output parameters to parameter connections; and (4) \textit{actual-out} vertices of the form \textit{in\_interface} = \textit{connection} to represent assignments that transfer return data of parameter connections to destination interfaces of the caller.

By means of these three operations, the architecture flow graph of the architectural design is generated. As an example, the AFG of sensor-to-actuator model (Fig. 2.1) is illustrated in Fig 5.2. A bold arc represents a component-internal control flow. The vertex $v$ of a component-internal control flow arc $v \rightarrow \text{bold} \ v'$ is called a direct predecessor of $v'$ and $v'$ a direct successor of $v$. A vertex can have zero, one, or two direct successors. A vertex $v$ with two direct successors represents a so called control expression constituting a Boolean condition. The two outgoing arcs of $v$ are attributed with \emph{T} for \textit{true} and \emph{F} for \textit{false} and correspond to the control flow in response to the condition evaluation. A bold dashed arc represents an interaction-based control flow due to the activation of a communication protocol. The execution of $v'$ of an arc $v \rightarrow \text{bold\_dash} \ v'$ coincides with the execution of $v$ according to the protocol. A thin arc represents a component-internal data flow. Finally, a thin dashed arc represents an inter-component data flow due to a data passing by value or by reference (shared data) protocol.
Figure 5.2: Architecture flow graph of the sensor-to-actuator model in Fig 2.1.
5.2 Verification Criteria

In order to accomplish research Goal 1, AQAF includes architecture-based verification criteria in terms of architectural control and data flows of AFGs, presented in paper A. In essence, there are component-internal control paths between the entry and the exit point of a component. If input interfaces are used and output interfaces are defined in the component-internal control flow paths, there exist component-internal data flow paths. There are inter-component paths between the exit point of a component and the entry point of another if their interfaces are connected. Finally, these path types may synthesize indirect paths between two components through one or several intermediate components. The defined verification criteria require all such relationships to be covered in the verification processes, to ensure completeness, consistency, and correctness of the architectural design or implementation.

A path is constrained if it is associated with property or requirement declarations, such as scheduling policies of processors, scheduling properties of tasks, protocols of shared resources, system modes, and minimum and maximum latencies and response times. A path in conjunction with the constraints of the path is referred to as a verification sequence by the verification criteria. A consistent architecture (design or implementation) is defined as an architecture in which all designed paths can be executed in compliance with the constraints. A complete architecture is defined as an architecture in which all paths can be executed by the specified input classes and an execution path is executed for every class of input. Correctness can only be verified if requirements are associated with the model, or if expected property declarations are considered as requirements. A correct architectural design is defined as an architecture in which all designed paths can be executed in compliance with the requirements. Under the assumption that there exists a searchable state space representation of the system architecture, these objectives are formalized as reachability problems by the verification criteria, which is in compliance with the principles of model checking and model-based test case generation.

The verification criteria presented in paper A have subsequently shown to lack considerations to a subset of path types that involve data flow sources and data flow sinks. As a result, faults pertaining to these types of execution paths may not be detectable by the defined verification criteria. In Section 5.2.1, the verification criteria in paper A are improved to include the excluded subset of path types.
5.2 Verification Criteria

5.2.1 Extension of Paper A: Inclusion of Data Flow Sources and Data Flow Sinks

An AFG contains different structural types of paths composed of control and data flows. Let $I = \{comp_1.i | comp_2.i\}$ denote the set of component interfaces of the architectural design. The set of interfaces may interact through the set of connections $C = \{c(s, d) \mid \text{source} s \in I \text{ and destination} d \in I\}$ and the set of behavioral models (BMs) of the architectural design. A BM refines the interaction of a connection if it operates on either the source or destination interface of the connection. Let $BMIR = \{bmir(i) \mid bmir(i)\}$ is a control flow path of a behavioral model that operates on $i \in I$ denote the set of BM interface refinements. A BM connects an input interface to an output interface in response to an operation on the input interface, i.e., that the operation on the output interface is control or data dependent on the input interface. Let $BMC = \{bmc(s, d) \mid bmc(s, d)\}$ is a control path of a behavioral model that operates on $d \in I$ in response to an operation on $s \in I$ denote the component interface connections connected through a BM. The possible types of AFG paths are then defined as:

**Definition 1.** $CIOP \subseteq I \times I$ is the set of Component-Internal Open-ended Paths from an interface $comp_1.i_1$ to another interface $comp_1.i_2$ such that $(comp_1.i_1, comp_1.i_2) \in CIOP$ iff $bmc(comp_1.i_1, comp_1.i_2) \in BMC$.

**Definition 2.** $CIOP \subseteq I$ is the set of Component-Internal Closed-ended Paths to or from an interface $comp_1.i_1$ such that $comp_1.i_1 \in CIOP$ iff $bmir(comp_1.i_1) \in BMC$ and $(comp_1.i_1, comp_1.i_2) \notin CIOP$ and $(comp_1.i_1, comp_1.i_2) \notin CIOP$.

**Definition 3.** $DCCP \subseteq I \times I$ is the set of Direct Component to Component Paths from an interface $comp_1.i_1$ to another interface $comp_2.i_2$ such that $(comp_1.i_1, comp_2.i_2) \in DCCP$ iff $bmir(comp_1.i_1) \in BMIR$ and $c(comp_1.i_1, comp_2.i_2) \in C$ and $bmir(comp_2.i_2) \in BMIR$.

**Definition 4.** $ICCP \subseteq I \times I \times I^*$ is the set of Indirect Component to Component Paths from an interface $comp_1.i_1$ to another interface $comp_2.i_2$ and is recursively defined to include three or more components. The base case is: $(comp_1.i_1, comp_2.i_2, t) \in ICCP$ iff $(comp_1.i_1, comp_2.i_2) \in DCCP$ and $(comp_2.i_2, comp_2.i_3) \in CIOP$ and $(comp_2.i_3, comp_2.i_4) \in DCCP$ and $t = (comp_1.i_1, comp_2.i_2, (comp_2.i_2, comp_2.i_3), (comp_2.i_3, comp_2.i_4))$. 
The inductive clause is: \( \langle \text{comp}_1.i_1, \text{comp}_x.i_y, t \rangle \in \text{ICCP} \) iff \( \langle \text{comp}_1.i_1, \text{comp}_2.i_2 \rangle \in \text{DCCP} \) and \( \langle \text{comp}_2.i_2, \text{comp}_2.i_3 \rangle \in \text{CIOP} \) and \( \langle \text{comp}_2.i_3, \text{comp}_x.i_y, t' \rangle \in \text{ICCP} \) and \( t = \langle \langle \text{comp}_1.i_1, \text{comp}_2.i_2 \rangle, \langle \text{comp}_2.i_2, \text{comp}_2.i_3 \rangle, \langle \text{comp}_2.i_3, t' \rangle \rangle \).

The path types are illustrated in Fig. 5.3. There is a component-internal open-ended path from \( \text{comp}_2.i_1 \) to \( \text{comp}_2.i_2 \) as they are connected through \( bmir_1 \), a control path of \( \text{comp}_2 \) that is guarded by \( \text{comp}_2.i_1 \) and creates a data flow to \( \text{comp}_2.i_2 \) in response to a true evaluation of the guard. Similarly, there is a component-internal open-ended path from \( \text{comp}_2.i_1 \) to \( \text{comp}_2.i_2 \) through \( bmir_2 \), a control path of \( \text{comp}_2 \) that is guarded by \( \text{comp}_2.i_1 \) and creates a data flow to \( \text{comp}_2.i_3 \) in response to a false evaluation of the guard. There are three component-internal closed-ended paths: one to \( \text{comp}_1.i_1 \) through \( bmir_1 \), a control path of \( \text{comp}_1 \) that defines \( \text{comp}_1.i_1 \); one from \( \text{comp}_2.i_1 \) through \( bmir_2 \), a control path of \( \text{comp}_3 \) that uses \( \text{comp}_3.i_1 \); and one from \( \text{comp}_4.i_1 \) through \( bmir_3 \), a control path of \( \text{comp}_4 \) that is guarded by \( \text{comp}_4.i_1 \). There are three indirect component to component paths: one to \( \text{comp}_1.i_1 \) through \( bmir_1 \), \( bmir_2 \), and \( bmir_3 \) indirectly connect \( \text{comp}_1.i_1 \) to \( \text{comp}_3.i_1 \) and where \( bmir_1 \), \( bmir_2 \), \( bmir_3 \) directly connect \( \text{comp}_1.i_1 \) to \( \text{comp}_4.i_1 \).

Figure 5.3: Illustration of control and data flow path types.

### 5.3 Formal Semantics

In order to formalize the semantics of AADL and accomplish research Goal 2, AQAF includes a set of transformation rules, defined in paper B, from AADL to UPPAAL timed automata. UPPAAL timed automata is selected due to its ability to express AADL behavior models, scheduling properties and synchroniza-
tions, and real-time properties, and due to the maturity of the UPPAAL model-checker. Through the transformation rules, AADL models may be subjected to the model-checker such that the designs may be verified through model checking. In addition, the model-checker is capable of producing traces of state space searches, which can be utilized for automated test case generation.

The transformation rules formalize schedulers and dispatchers of processor components, thread scheduling and execution states, behavior models of components, and component interactions. The transformation rules are tailored for architectural descriptions with synchronous, fixed-priority preemptive or non-preemptive execution models, as these commonly are used in dependable systems. Principles of modularization and encapsulation are however used to facilitate extensions such that other types of execution models can be incorporated.

The transformation process essentially maps an AADL model to a network of timed automata, where each processor component is mapped to a scheduler automaton, each thread component to a schedulable automaton, each subprogram component to an automaton callable through synchronization channels, and each port connection and shared data component to a global variable. A scheduler automaton controls the transition of thread scheduling states, from dispatches to completions, and of preemptions and context switches through synchronization channels. The “running” state of thread and subprogram automata constitute the AADL behavioral model state machine, however where behavioral model transitions that include timing properties, remote subprogram calls, or accesses to shared resources are complemented with constructs for potential context switches in the transformation to timed automata. An intermediate location is additionally added for timed behavioral model transitions wherein the corresponding time must progress before the target location is reached, as time can only progress between transitions in timed automata. For example, the output of mapping the behavior model transition $s_1 \rightarrow [\text{input}_1 > \text{input}_2] \rightarrow s_2 \{\text{computation}(20\text{ms}); \text{subprogram.call!}\}$ to timed automata is illustrated in Fig. 5.4.
The transformation process defined in paper B is not compatible with architectures that include protocols of shared resources. In Section 5.3.1, the transformation to timed automata is extended with constructs for protocols of shared resources.

5.3.1 Extension of Paper B: Inclusion of Shared Resources

A thread automaton, in its most basic form as illustrated in Fig. 5.5, consists of four locations: \textit{Awaiting Dispatch}, \textit{Ready}, \textit{Running}, and (if the thread operates on shared resources) \textit{Awaiting Resource}. Each thread is initially in the \textit{Awaiting Dispatch} location. An edge is then fired to the \textit{ready} location depending on the dispatch protocol. For periodic threads, the time of dispatch is entirely dependent on the clock in relation to the period of the thread. At the time of dispatch, data input connections are assigned to input ports of the thread, where ports of threads are mapped to local variables of the corresponding thread automaton. These assignments correspond to actual-in vertices of port connections in the AFG. Threads in the \textit{Ready} location are assigned to be executed by the processor component they are bound to according to a scheduling policy property. Assuming a scheduler with fixed priority preemptive scheduling policy, the thread with the highest priority is selected to run on the processor and thus transits to the \textit{Running} location. No more than one thread (per processing unit) is allowed to be in a \textit{Running} location simultaneously. A thread that operates on a shared resource accesses and releases it through a \textit{Get Resource()} and \textit{Release Resource()} service calls. If the semaphore of the resource already is locked at the time of access attempt, the thread transits to the \textit{Awaiting Resource} location.

A running thread is preempted and transits back to the ready location if
Figure 5.5: UPPAAL template of AADL threads. The behavior model (BM) of a thread replaces the "Running" location.

another thread with higher priority enters the Ready location. A thread in the Running location that completes its execution transits to the Awaiting Dispatch location. Output is simultaneously assigned to connections. These correspond to formal-in vertices of port connections of the AFG.

The automaton template for processor components that extends the original definition in paper B with mechanisms for shared resources is presented in Fig. 5.6. The labels of the scheduling automaton are defined as follows:

- (int)ready_queue[x]: is a sorted queue of currently dispatched threads. The queue is sorted according to a given scheduling policy where the first element in the queue (x=0) is the (identifier of the) thread being processed and where the second element is the next thread to be processed, and so forth.

- (clock)sch_clocks[x][2]: is a list of clocks in sets of two, each set referenced by an identifier x of a currently dispatched thread. Each dispatched thread has two clocks, the first (sch_clocks[x][0]) of thread with identifier x is used to keep track of a thread’s execution time, and the second (sch_clocks[x][1]) of thread with identifier x is used to keep track of a thread’s deadline.

- (int)sch_info[x][3]: is a list of threads’ scheduling properties (integers)
in sets of three, each set referenced by an identifier x of a currently dispatched thread. Each dispatched thread has three scheduling properties, the first (sch_info[x][0] of thread with identifier x) is the execution time, the second (sch_info[x][1] of thread with identifier x) is the deadline, and the third (sch_info[x][2] of thread with identifier x) is the priority.

- (int)preempt_stack[x][2]: is a stack of sets of currently preempted threads (integer identifiers) and the amount of time each thread has been preempted. Given a stack of preempted threads, the first set of elements in the stack (preempt_stack[0][0] is the thread identifier and preempt_stack[0][1] is the amount of time) corresponds to the thread that first was preempted.

- (int)nr_preempted: number of currently preempted threads.

- (int)threads: number of currently dispatched threads.

- (int)check_preempt: holds the identity of a thread that is dispatched at the same time as another thread is running. It is used to check if the dispatched thread preempts the running thread.

- (chan)dispatched[(int)x], (chan)run[(int)x], (chan)complete[(int)x], (chan)preempt[(int)x], (chan)blocked[(int)x], (chan)unblocked[(int)x]: are channels used to synchronize every thread transition of every thread in the system. Synchronization with a particular thread is done through its identity. For example, run[2] is a synchronization channel for the thread with identity “2”.

- (void)schprotocol((int)x): is a function sorting threads in the ready_queue according to a given scheduling policy. The function is called each time a thread dispatches where the thread’s identity is given as argument to the function.

- (void)completion((int)x): is a function removing threads from the ready_queue. The function is called each time a thread completes its execution, where the thread’s identity is given as argument to the function.

- (void)remove((int)x): is a function removing threads from the ready_queue. The function is called when a thread is blocked due to shared resources.
• (void)addTime(): is a function adding preempted time to the threads in the preempt_stack. The function is called when a preemption occurs, whereupon the execution time of the thread causing the preemption is added to the preemption time of every preempted thread.

• (void)removeTime(): is a function removing preempted time from the threads in the preempt stack. The function is called when a block due to shared resources occurs, whereupon the execution time of the thread is removed from the preemption time of every preempted thread.

• (void)checkTime((int)x): is a function adding preempted time to the threads in (int)preempt_stack[x][2]. The function is called when a thread dispatch not causing any preemption occurs, to check if the dispatched thread is prior to any preempted threads in the ready_queue whereupon preemption time is added.

The scheduler automaton includes two clocks per thread that is allocated to it. The reason for having two clocks per thread is that the UPPAAL language does only allow reset and comparison of clocks, i.e., clocks cannot be read or assigned. Because of these constraints, the time of completion of a preempted thread cannot be obtained solely from its execution time. In order to model thread preemption, a method considering the execution time of the threads causing preemption is used to calculate the time of completion of preempted threads. The method is illustrated in Fig. 5.7. $t_1, t_2$ and $t_3$ are denotations for threads where priority of $t_1 <$ priority of $t_2 <$ priority of $t_3$. $C_1$ is the execution time of $t_1$ and $D_1$ is the deadline. $c_1$ (corresponds to sch_clocks[i][0]) and $d_1$ (corresponds to sch_clocks[i][1]) are clocks for $t_1$, which are used to measure the time of completion and the time of a missed deadline respectively. $r_A$ is a variable used to summarize the time required to complete thread $t_1$ and all – during the execution of $t_1$ – dispatched threads with priorities higher than $t_1$. As shown in the illustration, the time of completion for $t_1$ is when the comparison $c_1$ equals $r_A$. In addition, $d_1 > D_1$ should not evaluate to true before the completion of $t_1$. The comparison is used for schedulability analysis where an evaluation to true indicates a missed deadline. Note that we are illustrating the method explicitly for thread $t_1$ although the methodology is applied to each thread. Furthermore, the behavior of the scheduler assumes immediate switching time of threads. If the processor the threads are bound to is specified with a Thread_Swap_Execution_Time property, the scheduler is extended with intermediate locations delaying the switching-time according to the specified property.
Figure 5.6: The scheduler automaton template.
The scheduler is initially in the Empty location when the system has been initialized. When dispatch occurs, the scheduler transits to the Schedule1 location whereby the corresponding thread is added to the ready_queue (via \textit{schprotocol}()) and its deadline clock is reset (corresponds to $d_1 = 0$ in Fig. 5.7). The Schedule1 location is a committed repetition of the Empty location, allowing several threads to be dispatched simultaneously through the edge labeled with channel \textit{dispatched}. The other edge to Schedule1 itself, labeled with channel \textit{unblocked}, allows for unblocking of threads. However, the edge can only be fired in response to a completion of a thread whereupon the availability of resources are checked. Succeeding to all simultaneous dispatches, the scheduler synchronizes with the first thread in the ready_queue and transits to the Running location through one of two different edges depending on which action should be executed. If the number of preempted threads is zero, or if the number is more than zero and the latest preempted thread is not the first in the ready_queue, the execution time clock of the thread to be run is reset (corresponds to $c_1 = 0$). If the number of preempted threads are more than zero and the latest preempted thread is the first thread in the ready_queue, the scheduler transits to the Running location without resetting its execution time clock since it already has been reset (corresponds to the start of execution of $t_1$ after preemption by $t_2$ and $t_3$). The scheduler remains in the Running location until the running thread gets blocked due to shared resources, until the running thread completes its execution, until another thread is dispatched, or until the running thread misses its deadline. A running thread that gets blocked due to shared resources synchronizes with the scheduler back to Schedule1, whereby the blocked thread is removed from the ready_queue and preemption time of

- $t_3$ \quad $R_1=C_1+C_2+C_3$
- $t_2$ \quad $R_1=C_1+C_2$
- $t_1$ \quad $c_1=0$, $d_1=0$, $R_1=C_1$
- $c_1=R_1$, $d_1>D_1$?

Figure 5.7: Thread execution schema for threads $t_1$, $t_2$ and $t_3$, where $\uparrow$ indicates dispatch and $\downarrow$ indicates completion.
possible preempted threads is adjusted with respect to the execution time of the blocked thread. If a running thread completes its execution (corresponds to \(c_A = r_A\)), the scheduler transits to the LookUp location through one of two different edges. Note that the running location is modeled with an invariant in order to force a fire of the completion edge at the time of completion. The two edges have guards for execution time where additional expressions are used to differentiate between a preempted thread and a thread which has not been preempted. If the thread has not been preempted, the thread is simply removed from the ready_queue (through the completion() function). A preempted thread, on the other hand, is not only removed from the ready_queue, but also from the preempt_stack. From the LookUp location, the scheduler transits to the Empty or Schedule1 location depending on whether there are any dispatched threads left or not. If dispatched threads still exist, the scheduler synchronizes with possibly blocked threads to check the availability of shared resources in response to the thread completion. This allows for unblocking of threads when the scheduler enters Schedule1.

If a dispatch occurs when the scheduler is in the Running location, an edge is fired to the Schedule2 location, whereupon the thread is added to the ready_queue and the corresponding execution time clock is reset. Three different edges are available from the Schedule2 location depending on if the recently dispatched thread was scheduled as the first thread in the ready_queue or not. If scheduled as the first thread in the ready_queue, that is, if it preempts the previously running thread, the scheduler transits to the Preemption location through one of the two edges depending on whether the preempted thread already exist in the preempt_stack or not. Whereby the edges from the Schedule2 location to the Preemption location, the preempted thread is added to the preempt_stack if it previously has not been preempted, and preempted time is added – to all preempted threads – through the addTime() function (corresponds to \(r_A = C_A + C_B\) or \(r_A = C_A + C_B + C_C\)). On the other hand, if the recently dispatched thread does not cause a preemption, no further actions are taken other than adding preempted time – if the thread is scheduled prior to currently preempted threads – to preempted threads through the checkTime() function. From the Preemption location, the scheduler synchronizes with the first thread in the ready_queue for execution. Note that the Preemption location has an edge to itself to allow simultaneous dispatches of threads.

The edge from the Running location to the MissedDeadline location is modeled for schedulability analysis. The Running location is modeled with an invariant that forces a fire of the edge whenever the running thread misses its deadline (corresponds to \(d_A > D_A\)). Thus, any internal, direct, or indirect
5.4 Model Checking with Observer Automata

In order to formally and automatically perform verification of architectural designs according to the verification criteria, and partly accomplish research Goal 3, AQAF generates so called observer automata based on verification sequences (paths+constraints) extracted from AFGs. The theory of observers within AQAF is presented in paper D. Observers have been developed to provide a flexible method for specifying coverage criteria for model checking and test case generation [77]. The execution is formulated as a reachability problem, which conforms to the verification criteria of AQAF.

An observer \( \langle O, o_0, o_{\text{accept}}, E_{\text{obs}} \rangle \) has a set of observer locations \( O \), an initial observer location \( o_0 \in O \), a final “acceptance” location \( o_{\text{accept}} \in O \), and a set of observer edges \( E_{\text{obs}} \) on the form \( o \xrightarrow{g,a,u} o' \), which predicates \( g \) and \( a \) are dependent on the elements of the timed automata model the observer is supposed to monitor. The observer edges will thereby be fired when the observed elements are executed, eventually leading to the acceptance location when the coverage criterion has been achieved. The coverage criterion of a verification sequence is the corresponding path in the timed automata model of the architectural design. Each control flow arc in an AFG path corresponds to the firing of one particular edge in the timed automata model and each data flow arc to the firings of two edges in a sequence, one where the variable is defined and one where it is used. Consequently, an observer automaton is created for each verification sequence by creating an observer edge, or a sequence of two observer edges, for each arc in the AFG path. Moreover, observers are labelled with invariants, guards, actions, and clocks to specify the constraints in which the elements must be covered, such as maximum response times of components and latencies of connections. For example, if an observer shall observe a data flow through a connection that is associated with a maximum latency property of \( x \) milliseconds, the observer is equipped with a clock construct that logs the time from when data enters the connection until the data exits the connection. A guard of the form \( o \xrightarrow{cl=x,a,u} o' \) subsequently demands compliance with the timing requirement before the acceptance location can be reached. By structuring the observer edges according to the prescribed path, satisfiability of the verification sequence may be verified through a reachability formula of the form \( E \xrightarrow{<>} o_{\text{accept}} \) – meaning “there exists one path where \( o_{\text{accept}} \) eventually holds”. The verification sequence passes if the model satisfies the formula.
5.5 Model-based Testing with Observer Automata

To finalize the accomplishment of research Goal 3, test cases that test the conformance of an implementation to its model must be generated according to the verification criteria. A satisfied observer generates a trace \( \langle \ell_0, \phi_0, \sigma_0 \rangle \xrightarrow{a_1/d_1} \langle \ell_1, \phi_1, \sigma_1 \rangle \xrightarrow{a_2/d_2} \cdots \xrightarrow{a_n/d_n} \langle \ell_n, \phi_n, \sigma_n \rangle \) that contains information about the initial state of the system and its environment before the path is executed, the input or the sequence of inputs needed to stimulate an execution of the path, and the expected output or sequence of outputs. In addition, the trace contains information on the timing of inputs and the expected outputs. Consequently, each observer trace holds the necessary information to test whether a corresponding behavior exists within the implementation. AQAF includes an algorithm, defined in paper D, that converts the generated observer traces into test cases, such that the conformance of the implementation with respect to its intended design can be verified.

5.6 Selective Regression Verification

In order to accomplish research Goal 4, AQAF includes a selective regression verification technique, presented in paper C, that only re-executes those verification sequences that may be impacted by the modification. The technique is essentially composed of four consecutive stages. The first stage identifies the change. An architectural design change will be reflected in differences between the AFGs of the initial and changed model, where a comparison establishes the identification. The second stage identifies the other parts of the modified architectural design that possibly are impacted by the change. Impact analysis is performed through forward slicing of the changed model with respect to the change. The slicing technique provided by AQAF calculates the forward slice based on the control and data dependencies of the model. A data flow implies a data dependency whereas control dependencies are calculated through post-domination analysis of the component-internal control flows of the AFG. Assume that \( v_x, v_y, \) and \( v_z \) are non-actual in/out and non-formal in/out vertices and contained within the same component. A vertex \( v_x \) is post-dominated by a vertex \( v_y \) if every path \( P = v_1 \xrightarrow{c} v_2 \xrightarrow{c} \cdots \xrightarrow{c} v_n \) from \( v_x \) to the exit vertex, i.e. \( v_1 = v_x \) and \( v_n \) is the exit vertex, includes \( v_y \), where \( v \xrightarrow{c} v' \) denotes a component-internal control flow. Control dependence is then defined as:
**Definition 5.** A vertex $v_y$ is control dependent on a vertex $v_x$ iff 1) $v_x$ is an entry vertex and $v_y$ is not nested within any loop or conditional vertex, or 2) there exists a path $P = v_1 \rightarrow_c v_2 \rightarrow_c \cdots \rightarrow_c v_n$ from $v_x$ to $v_y$ such that any vertex $v_z$ in $P$ is post-dominated by $v_y$, and $v_x$ is not post-dominated by $v_y$ ($v_x$ must be a control expression).

An illustration of the control dependencies of *Logics* in the running example is presented Fig. 5.8, where an arc $v \rightarrow v'$ denotes that $v'$ is control dependent on $v$. Regarding constructs of vertices that represent component connections, interaction-based control flows and calls of the AFG are themselves control dependencies since the source vertex initiates the execution of the target vertex.

![Figure 5.8: Control dependencies of Logics in Fig. 2.1.](attachment:image.png)

The resultant control and data dependencies are merged into a single graph referred to as the architecture dependence graph (ADG), through which the slice can be quickly determined by searching for all vertices that are forward-reachable from the changed or new vertices through the arcs. The third stage is to select only those verification sequences that cover vertices in the slice for re-verification.

The proposed slicing technique is static in the sense that the slice is calculated without any information from an actual execution of the system. The set of affected verification sequences may be reduced by inter-observer satisfiability independence analysis, which adds dynamical (chronological) dependencies to the selection process. If an observer $obs_x$ (verification sequence) may be satisfied without satisfying another observer $obs_y$, then $obs_x$ is satisfiable independently from the path observed by $obs_y$. A previously satisfied...
observer, which satisfiability is independent to each observer that covers the change (excluding the potential impact), will also be satisfiable in the reverification process since the path it observes may be executed prior to an execution of the changed elements. The verification sequence can therefore be deselected even if it covers a vertex in the static forward slice. The fourth and last stage is therefore to deselect all verification sequences that do not cover a possibly impacted part according to the dynamic dependencies, to generate the minimal subset of the regression verification suite and execute it.

The dynamic dependence analysis is introduced in Paper D. However, the analysis occasionally proved to be incomplete in the initial instances of the evaluation study presented in paper F. Results showed that the analysis lacked considerations to conditional dynamic dependencies, which had to be solved before satisfactory results could be obtained. In Section 5.6.1, the dynamic dependence analysis introduced in Paper D is extended with considerations to conditional dependencies.

### 5.6.1 Extension of Paper D: Dynamic Dependence Analysis

The dynamic dependence analysis is performed between verification cycles through satisfiability checking of formulae on the form: $E <> ObserverX.Acceptance$ and not $ObserverY.Acceptance$. If the formula is satisfied, the path observed by $ObserverX$ may execute independently from the path observed by $ObserverY$. If the formula is unsatisfied, the path observed by $ObserverX$ is chronologically dependent on the path observed by $ObserverY$, i.e., an execution of path “$X$” must be preceded by an execution of path “$Y$”. Note that the analysis evaluates chronological dependencies without regard to their causes. Since chronological dependencies may be solely caused by scheduling properties, an identified dynamical dependency does not imply that a path is control or data dependent on another path. Moreover, given three paths $X$, $Y$, and $Z$, a satisfiability of (i) “$E <> ObserverX.Acceptance$ and not $ObserverY.Acceptance$” and of (ii) “$E <> ObserverX.Acceptance$ and not $ObserverZ.Acceptance$” does not imply that (iii) “$E <> ObserverX.Acceptance$ and not $ObserverY.Acceptance$ and not $ObserverZ.Acceptance$” is satisfiable.

In other words, the dynamical independence of path $X$ with respect to paths $Y$ and $Z$ may be conditional. For example, presume a system with two unconnected and periodically dispatched threads, $Thread_1$ and $Thread_2$, that have equal periods and where $Thread_2$ is prioritized over $Thread_1$. Further assume there exist one path, $X$, through $Thread_1$ and two paths, $Y$ and $Z$, through $Thread_2$ and that either $Y$ or $Z$ may execute in each dispatch of $Thread_2$.
depending on the input. Due to the scheduling properties, an execution of $X$ ($Thread_1$) is always preceded by an execution of either $Y$ or $Z$ ($Thread_2$). However, $X$ must not be preceded by $Y$ as long as $Z$ precedes $X$, and $X$ must not be preceded by $Z$ as long as $Y$ precedes $X$. In this case, where (i) and (ii) are satisfiable but (iii) is unsatisfiable, the satisfiability of $ObserverX$ is conditionally independent to $ObserverY$ and $ObserverZ$. Note that conditional dynamic dependencies may be caused by control and data dependencies, irrespectively of scheduling properties. If $Thread_2$ is connected to $Thread_1$ such that $X$ only is executed when input is available from an execution by either $Y$ or $Z$ of $Thread_2$ (and some other path through $Thread_1$ otherwise), then $X$ would dynamically be conditionally independent to $Y$ and $Z$ even when $Thread_1$ is prioritized over $Thread_2$.

An analysis of conditional dependencies is necessary for dynamical impact analyses of changes that modify several paths. A change to either $Y$ or $Z$ in the above example cannot impact the satisfiability of $ObserverX$, but a change to both might. In case of an unconditional independence of $X$ with respect to $Y$ and $Z$, where (i), (ii), and (iii) are satisfiable, neither a change to $Y$, $Z$, or $Y$ and $Z$ can impact the satisfiability of $ObserverX$.

In theory, the worst case number of formulae that must be evaluated to be able to determine the impact of a change to a single component or connection is:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (x_i \cdot 2^{x_j - 1} - 1) + \sum_{i=1}^{n} (Y \cdot 2^{x_i} - 1) + (X \cdot Y)$$

where $n$ is the number of software components in the architecture, $x_i$ is the number of component-internal paths of software component $i$, $X$ is the total number of component-internal paths, and $Y$ is the total number of inter-component paths (software component connections). In other words, the satisfiability of (1) each component-internal path with respect to each subset of other paths pertaining to a component (excluding the empty set), (2) each inter-component path with respect to each subset of other paths pertaining to a component (excluding the empty set), and (3) each component-internal path with respect to each inter-component path, must be evaluated. If the change may constitute changes to any number of components and connections, the number of formulae to evaluate increases to $n \cdot 2^{n-1} - n$, where $n$ is the total number of component-internal and inter-component paths of the architectural design. Depending on the structure of the design, the law of transitivity and a suitable order of formulae evaluations may significantly reduce the necessary number of formulae to evaluate. In the best case, each path is unconditionally inde-
dependent to all other paths, where an evaluation of $n$ formulae is enough to determine all possible dynamical dependencies of the architectural design. Such a structure can however only be achieved by a completely parallel and synchronized execution of tasks that have no control or data dependencies among them. Another precondition for reducing the number of formula evaluations for this type of independent structure is that the analysis evaluates dependencies to the largest possible sets first. Let \( \{ \text{path}_1, \text{path}_2, \text{path}_3, \ldots, \text{path}_n \} \) denote the set of paths of an architectural design and \( \{ \text{obs}_1, \text{obs}_2, \text{obs}_3, \ldots, \text{obs}_n \} \) the corresponding observers. The following formulae evaluate the dynamic independence of each path with respect to the largest possible path set:

\[
E \leftrightarrow \text{obs}_1 \land \neg \text{obs}_2 \land \neg \text{obs}_3 \ldots \land \neg \text{obs}_n
\]

\[
E \leftrightarrow \neg \text{obs}_1 \land \text{obs}_2 \land \neg \text{obs}_3 \ldots \land \neg \text{obs}_n
\]

\[
E \leftrightarrow \neg \text{obs}_1 \land \neg \text{obs}_2 \land \text{obs}_3 \ldots \land \neg \text{obs}_n
\]

\[
E \leftrightarrow \neg \text{obs}_1 \land \neg \text{obs}_2 \land \neg \text{obs}_3 \ldots \land \text{obs}_n
\]

If all paths are unconditionally independent to all other paths, then the above formulae are satisfied. This implies that any formula that contains a subset of the negated observers also is satisfiable: if \( \text{obs}_1 \land \neg \text{obs}_2 \land \neg \text{obs}_3 \ldots \land \neg \text{obs}_n \) is satisfiable then \( \text{obs}_1 \land \neg \text{obs}_2 \land \neg \text{obs}_3 \ldots \land \neg \text{obs}_{n-1} \) must be satisfiable as well. Hence, the analysis is in this case completed after \( n \) formula evaluations, given that the largest sets are evaluated first. On the other hand, if the smallest sets are evaluated first, the necessary number of formula to evaluate would be \( n \ast 2^{n-1} - n \) to determine all possible independencies between the paths. The largest sets first strategy becomes inefficient if the dependence vs. independence ratio is inverted, e.g. where the vast majority of paths are dynamically dependent on a single path, where a smallest sets first strategy is most efficient. However, the analysis is in general resource consuming and increases exponentially by the number of execution paths. Nevertheless, the analysis is implemented in AQAT such that the analysis is performed in the background between verification cycles – a combination of smallest sets first and largest sets first strategies is implemented. Consequently, the additional overhead has little effect on the active regression verification process and is not experienced by the tool user. The analysis must neither fully complete to be usable and can be interrupted at any time to initiate a selective regression verification process. The capability of deselecting unnecessary verification sequences to re-execute is however increasing by the degree to which the analysis is completed.
The analyzed dependencies of the architectural design may also be used for hazard analysis, where potential error propagations, common cause failures, single point failures, etc., may be deduced from the information. Furthermore, the analyzed dependencies may be used for reusability analysis, parallelization of independent functions, and for the analysis of potential hazardous relations between critical and non-critical functionality in mixed criticality systems.

5.7 The Architecture Quality Assurance Tool

A computer tool referred to as the Architecture Quality Assurance Tool (AQAT) – available for use at http://www.idt.mdh.se/~AQAT/ – is developed based on the theory of AQAF, to enable an easy adoption into industrial practices. AQAT, presented in paper E, is a Java application with a graphical user interface (GUI). This section demonstrates the main cases in which the tool may be used with respect to the GUI. The reader is referred to paper E for a presentation of back-end implementation details.

![AQAT](image)

Figure 5.9: GUI: create new project.
The tool is compatible with AADL models stored as XML files, which may be generated by the standard AADL editors. The tool utilizes the standalone UPPAAL model-checker Verifyta [38] for model checking, test case generation, and inter-observer satisfiability independence analysis. In addition to model checking and model-based testing of control and data flow paths, the tool is equipped with a feature for schedulability analysis. Furthermore, the tool allows the user to open the transformed AADL model (the timed automata model) within the UPPAAL environment, where the behavior of the AADL model may be simulated, inspected, and subjected to customized model checking.

Figure 5.10: GUI: select verification options.

The tool manages verification projects through AQAT project files. A new project is created through File→New Project in the menu bar, as illustrated in Fig. 5.9. The user then selects the AADL model under analysis, and possibly a previously saved AQAT project pertaining to the verification of a prior version of the model. The selective regression verification feature is engaged by the latter action. In this demonstration of AQAT, the sensor-to-actuator AADL model presented in Section 2.1 is used.

The user is subsequently given the option to configure the verification pro-
The Architecture Quality Assurance Tool

Figure 5.11: GUI: select component(s) that represents the behavior of the system environment.

cess according to case-specific needs, as presented in Fig. 5.10. By default, the tool will not perform verification of, or generate test cases from, control and data flow paths that include component-internal or inter-component loops. If the user needs to include potential loops in the analysis, the user may set the maximum number of loops the tool should consider. These bounds do not have any effect on the verification of the sensor-to-actuator example since it is free from loops. The user may also restrict the state space search for model checking and test suite generation by selecting an upper bound of clock cycles to analyze, with intervals corresponding to the duration of one system hyper-period – a value of zero entails no bound. Although the ability to detect faults that exist beyond these limits is lost by these restrictions, they might be necessary to set for large and complex systems in order to conclude the verification processes within a reasonable amount of time. The user may finally request a depth first state space search instead of the default breadth first search strategy, a schedulability analysis, a test suite generation, and/or a record of architectural dependencies to enable a selective regression verification process in a possible future reverification project. If a test suite generation is requested, an additional dialogue box is displayed wherein the user is asked to mark the components of the model that represent the possible behaviors of the system environment, as presented in Fig. 5.11. The information is necessary for the tool to identify the input interfaces/connections of the model that will constitute the controlled interfaces of the system under test. Test cases shall stimulate the implementation in place of Sensor in this case.

The tool initiates the verification process when the options have been selected. The progress of the process is displayed in real time through the main
window of the GUI, as presented in Fig. 5.12. The left-hand window pane displays the status of major framework processes that are executing and their key results. Verdicts from the model checking process are displayed through green, red, and a yellow symbols, depending on if a path is executable (green circle), unexecutable (red rectangle), or if the executability is inconclusive (yellow circle). An inconclusive verdict indicates that the model-checker ran out of memory in the model checking process. With respect to the sensor-to-actuator model, two faults are detected by the tool. First, the minimum latency property of connection control (1 ms) is exceeded in the second dispatch of Actuator as the dispatch coincides with the completion of Logics (all tasks are simultane-
ously released at system initialization). The time from when output is produced by Logics until it is read by Actuator will consequently be below the required minimum. Second, the control signal to Actuator cannot be higher than three according to the modeled input range and computations. The corresponding predicate and control flow within Actuator is consequently unreachable.

Figure 5.13: GUI: open results – model checking.

The right-hand window pane displays details from back-end processes and messages that are sent from the model-checker. Subsequent to the completion of the model checking process, and the test suite generation process if selected, the user may open a comprehensive description of the project and the results through the menu bar, including the extracted verification sequences/paths and their verdicts, the generated test suite, a summary of the project, and the inter-observer satisfiability independences, as illustrated in Fig. 5.13–5.16.

The user may also open the AADL model within the UPPAAL environment.
as illustrated in Fig. 5.17. In case of an unexecutable path verdict, the tool displays a path-specific launch button together with the warning through which the particular faulty behavior can be debugged within the UpPAAL environment, as shown in Fig. 5.12.

When the verification process has completed, the user may save the project as illustrated in Fig. 5.18. If the architectural design subsequently is modified, e.g. in order to correct the detected faults of the running example, the user may refer to the previous verification project to selectively execute the regression verification process. In order to demonstrate this feature, the running example is modified to correct the two detected faults. For simplicity, the unachieved time constraint in the initial version is alleviated whereas the unreachable expressions are removed to correct the two faults. The second version of the sensor-to-actuator AADL model is subsequently opened in a new project, as il-
illustrated in Fig. 5.19, wherein the previous verification project simultaneously is referenced, as illustrated in Fig. 5.20. The tool will then, subsequent to the configuration of the verification project as presented in Fig. 5.10, execute the regression verification process selectively in response to an impact analysis of the modification, as illustrated in Fig. 5.21. The result report of the regression verification project is correspondingly extended with a “Regression” tab that presents the identified changes and their potential impact, as presented in Fig. 5.22.
Figure 5.16: GUI: open results – satisfiability independence.
Figure 5.17: GUI: open AADL model in the UPPAAL environment.
Figure 5.18: GUI: save verification project.

Figure 5.19: GUI: create (regression) verification project.
Figure 5.20: GUI: reference the previous verification project.

Figure 5.21: GUI: selective regression verification process.
Figure 5.22: GUI: open results – regression.
Chapter 6

Evaluation

6.1 Utility

The utility of AQAT and the underlying AQAF theory are evaluated by means of an industrial case study, presented in paper F, to accomplish research goal 5. The study comprises an application of AQAT to a safety-critical train control system. In order to ensure coverage of fault types and to produce a statistically significant data set from which effectiveness and efficiency can be reliably assessed, the method of fault injection is used as case study design.

6.1.1 Design of Study

Effectiveness of model checking and model-based testing is measured in terms of the ratio of detected faults to the number of injected faults. Efficiency, on the other hand, is measured in terms of time and memory consumption of the conducted verification processes. Furthermore, each fault injection may be treated as a modification of the default architectural design, where the effectiveness and efficiency of the selective regression verification technique is assessed by comparing the results with a re-run all approach. Effectiveness of the selective regression verification technique is thereby measured in terms of the ratio of unsatisfied non-selected observers to the number of selected and non-selected unsatisfied observers. A lower ratio denotes a higher effectiveness in this case. A ratio of zero implies that the selective regression verification technique does not deselect any verification sequence that reveals a fault (unsatisfied observer) in the modified architecture. Finally, efficiency is measured by the proportion
of consumed resources of the selective approach compared to the re-run all approach.

By using a common practice confidence level of 95% (95% of the area under a normal curve lies within roughly 1.96 standard deviations of the mean), a conservative expected standard deviation of 0.5, and a ± 5% margin of error, the number of fault injections is set to 385 ($1.96^2 \times 0.5 \times (1 - 0.5)/0.05^2$).

In other words, 385 design faults, 385 implementation faults, and 385 faulty changes are evaluated in the study. The set of fault types that are considered in the study are selected based on the set of AADL expressions that define and constrain the control and data flows of the architecture. The resultant types of faults are:

FT1: Absent, unachievable, or incorrect control expression (guard)

FT2: Absent or incorrect data assignment, event, or call (action)

FT3: Absent or incorrect port connection

FT4: Absent or incorrect parameter connection

FT5: Absent, incorrect, or incompatible timing property (e.g. excessive response time)

FT6: Absent, incorrect, or incompatible protocol or use of shared resource (deadlock, livelock, starvation, and priority inversion of threads)

FT7: Absent, incorrect, or incompatible scheduling property (missed deadline)

FT8: Absent behavior model transition

FT9: Absent or incorrect transition priority

For the purpose of the case study, AQAT is extended with a module that automatically performs the fault injections. The module essentially parses the AADL model and injects a fault upon the arrival of a selected expression, which subsequently constitutes a faulty design that also may be treated as a faulty version (change) of the initial design. The distribution of fault injections across fault types is thereby largely determined by the frequency of the corresponding AADL expressions. The timed automata models that are generated from the faulty AADL models are treated as faulty implementations of the initial fault-free architectural design, to validate the fault detection effectiveness.
of the test suite generation technique. The test suite is consequently generated from the default AADL model and executed against each faulty timed automata model. For the purpose of the case study design, each generated test case is scripted in timed automata such that they can be automatically executed. AQAT is extended in the study with a module that automatically performs the encoding.

In order to detect potential false positives and false negatives, i.e., to detect whether the tool falsely declares presence or absence of faults, the default fault-free version of the architectural design is subjected to verification. If the model checking technique is valid, the result must be satisfied observers. This serves as supporting evidence for the presence or absence of false positives and false negatives. Similarly, the timed automata model of the default fault-free architectural design is subjected to testing. If the test suite generation technique is valid and the implementation actually conforms to the model, the result of model-based testing must be passed test cases. Since the model evidently conforms to the model itself, the treatment of the generated timed automata model as an implementation serves as supporting evidence for the presence or absence of false positives and false negatives.

### 6.1.2 Results

The descriptive statistics of the results are presented in Table 6.1, which displays the averages, standard deviations, minimums, maximums, and the totals in five consecutive row sections. The first column of the table presents the fault type ("0" denotes no injected fault). The second column presents the number of extracted verification sequences for full coverage of the AADL model. The third column presents the number of selected verification sequences (for selective regression verification) when the fault injection is treated as a modification of the default model. The fourth column presents the number of unsatisfied selected verification sequences (observers) whereas the fifth column presents the total number of unsatisfied verification sequences (selected and deselected). Columns six to eleven present the time and memory consumption of model checking the selected set of verification sequences versus the complete set of verification sequences. Note that the model-based testing technique uses the results (traces) of model checking to generate the test suite. Hence, the resource consumption of model checking also indicates the resource consumption of test suite generation. The last column presents the number of failed test cases of the test suite generated from the default model.

Since the minimum number of unsatisfied observers over the complete sam-
ple of fault injections is larger than zero, 385 out of 385 design faults are detected. No implementation faults of FT5 were detected by the generated test suite. The invalidity is expected as the fault type is an inconsistent latency property, which in the model does not affect the execution but impose an analysis constraint on it. It is not sound to treat the faulty timed automata model as a faulty implementation in this case since the inconsistent property must be manifested in the execution to emulate an implementation fault. Additional five (the quantity is not deducible from Table 6.1) faults of FT1, FT3 and FT7 were not detected at the implementation level – the minimum number of failed test cases is zero for each type. Given that the unsound implementation faults of FT5 are disregarding, 338 out of 343 (385-42*FT5) implementation faults are detected. The results consequently suggest a 100% fault detection rate at the model level and a 98.5% fault detection rate at the implementation level.

On average, model checking of a faulty AADL model of the case subject consumed approximately 6.5 minutes and 2.4 GB of memory, whereas model checking of the fault-free model consumed approximately 4 minutes and 1.3 GB of memory. The results also indicate that faults may significantly reduce as well as increase the time consumption of model checking (and test case generation), depending on the type and the location of the fault. The minimum time consumption of the results is 15 seconds whereas the maximum is approximately 43 minutes. Faulty mechanisms of shared resources (FT6), faulty scheduling properties (FT7), missing transitions (FT8), and missing connections (FT3) caused the most comprehensive negative effect on the architectural design and implementation in terms of percentage of unsatisfied verification sequences and failed test cases.

The selective regression verification technique increased the performance of regression verification on average by 6.4% compared to a re-run all approach. A total of 13844 verification sequences were extracted in the study, 11168 of which were selected for regression verification. 5886 of the selected sequences were unsatisfiable compared to 5886 in the re-run all approach. The selective regression verification technique and the impact analysis technique are consequently effective in selecting verification sequences that cover impacted elements of a modified architectural design. The technique is in this study most efficient for changes of actions and transition priorities. However, the selective approach is not usable for changes of scheduling properties since they have no measurable impact on the AFG. Consequently, no slicing or selection can be performed in response to such changes.
Table 6.1: Utility Results

<table>
<thead>
<tr>
<th>Fault type</th>
<th>No. seq.</th>
<th>No. unsat.</th>
<th>No. obs.</th>
<th>Sel. (sec)</th>
<th>All (sec)</th>
<th>Mem. (%)</th>
<th>Time (%)</th>
<th>No. failed TCs (of 38)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVERAGE</td>
<td>38</td>
<td>28.1</td>
<td>27.6</td>
<td>25.3</td>
<td>25.2</td>
<td>36.1</td>
<td>34.3</td>
<td>33.2</td>
</tr>
<tr>
<td>DEVIATION</td>
<td>38</td>
<td>28.1</td>
<td>27.6</td>
<td>25.3</td>
<td>25.2</td>
<td>36.1</td>
<td>34.3</td>
<td>33.2</td>
</tr>
<tr>
<td>STD</td>
<td>38</td>
<td>28.1</td>
<td>27.6</td>
<td>25.3</td>
<td>25.2</td>
<td>36.1</td>
<td>34.3</td>
<td>33.2</td>
</tr>
</tbody>
</table>

6.1.3 Discussion

The tool is unable to detect 1.5% of the injected implementation faults in this study. The framework is unable to detect nonconformity of an implementation to its design specification in cases where the faulty implementation produces the expected output, i.e., in cases where the post-condition of a test is fulfilled but the behavior that produce it does not correspond to the architectural design. To detect nonconformity in such cases, an implementation instrumentation that measures code coverage must be implemented.

The resource consumption of model checking varies significantly in the study. The results suggest that there are primarily two parameters that determine the variation. Faulty mechanisms of shared resources (critical sections) produced the lowest average time consumption (38 seconds). The cause is a significantly reduced state space due to starvations, missed deadlines, and deadlocks of threads. On the other hand, faulty actions produced the highest average time consumption (925 seconds). The increase is however not cause by a significantly increased state space. The increase of necessary resources is caused by multiple unsatisfiable verification sequences in relation to relatively large state spaces. Unsatisfiability of verification sequences require a complete state space search, with all possible combinations of system inputs, which tend to be relatively time consuming.

The selective regression verification technique increased the performance of regression verification on average by 6.4% in this study. However, an absent behavior model transition on average reduces the number of prescribed paths by 39% (1 − 23.3/38), which suggests that the architectural design is composed of a rather interdependent structure. The efficiency of the selective regression verification technique is therefore likely to significantly increase for systems with less interdependent structures.

6.2 Scalability

The case subject utilized in the utility study corresponds to a moderate single-core case, which provides insufficient data for scalability evaluations. In order to assess the limitations of AQAT, the evaluation is extended with a study wherein AQAT is applied to a wide range of case subjects, including outlier cases.
6.2 Scalability

6.2.1 Design of Study

The essential external factors that affect scalability of AQAT and the underlying AQAF theory are the performance of the computer platform running the tool and the complexity of the architectural design under analysis. In addition, the utilized model checking environment of UPPAAL is based on a client-server architecture, where model-checkers may be remotely installed on multiple servers and communicate with clients through TCP/IP. The sequential execution of verification sequences (satisfiability checking of observers) and of dynamic dependence analyses (inter-observer satisfiability checking) by AQAT, which have been shown to be rather resource consuming, may consequently be completely parallelized in a distributed installation. The capability of AQAT to handle a growing system complexity may therefore be increased simply through material investments in the installation. However, we intend to evaluate the scalability of AQAT under the default conditions, i.e., centralized computing on a standard personal computer, to focus on the identification of potential methodological vulnerabilities. Evaluations are thereby conducted for a variety of architectural design complexities upon a fixed default installation configuration.

The main parameters that impact the scalability of AQAT in terms of system complexity are the number of processes (threads), the number of processors, and the cyclomatic complexity of the architectural design under analysis. The questions we investigate are the degrees to which AQAT can accommodate increasing work loads in these dimensions:

1. How do an increasing cyclomatic complexity, number of processes (threads), and number of processors impact the performance of AQAT?
2. What are the maximum cyclomatic complexity, number of processes, and number of processors AQAT is capable to verify?
3. Which framework parts are most vulnerable to an increasing cyclomatic complexity, number of processes, and number of processors?

In order to rigorously assess scalability limitations the study encompasses mock-up cases ranging from single-core and single-threading systems to multi-core and multi-threading systems. The approach is to initiate the evaluation with simplistic architectural designs and incrementally add threads, processor cores, and cyclomatic complexity (execution paths) until an excessive time or memory consumption is reached. Both breadth-first and depth-first state space search strategies are utilized in the study. However, only the result from the...
most efficient configuration for each case subject is presented. Moreover, the
design of threads is selected differently in limitation evaluations with respect
to cyclomatic complexity compared to evaluations with respect to quantities
of threads and processors. In the former limitation evaluations, each thread is
uniformly designed with two possible (component-internal) execution paths.
In the latter, threads are uniformly designed with a single execution path. By
selecting the simplest possible design for threads in the latter limitation studies,
the maximum number of threads and processing cores AQAT may accommo-
date is assessed. In all cases, the threads are connected according to a pipeline
design pattern as presented in Fig. 6.1 and 6.2, where each thread (except for
the first) has one input interface connected to the preceding thread and one
output interface connected to the succeeding thread.

Figure 6.1: Pipeline pattern for evaluations with respect to quantities of pro-
cesses and processing cores.

Figure 6.2: Pipeline pattern for evaluations with respect to cyclomatic com-
plexity.

A pipeline design pattern is selected to minimize the impact of scheduling
properties and context switches on the overall system complexity such that the
upper limits of cores, threads, and cyclomatic complexity can be evaluated. In
architectural designs with multiple cores, a uniform and independent pipeline
of threads is allocated to each core. For example, a system with a dual core
processor and ten threads implies two identical and independent five-threaded
pipelines, each of which is executed in parallel by a single core. Moreover,
each execution path within the threads simply assigns the input data to the out-
put interface. Scheduling properties are set such that the threads execute ac-
cording to the pipeline sequence, i.e., output data produced by the initial thread
6.2 Scalability

reaches the final thread when each thread has executed once. In this manner, the number of paths grows exponentially when threads are added in limitation evaluations with respect to cyclomatic complexity (Fig. 6.2) and linearly otherwise (Fig. 6.1).

We define cyclomatic complexity in this study as the number of end to end paths. The cyclomatic complexity (CC) is thereby constantly one in pipelines of single-path threads regardless of the number of threads in the pipeline. In pipelines with \( n \) dual-path threads, the cyclomatic complexity is \( n^2 \).

For each application of AQAT in this study the coverage criteria is set such that each component-internal, direct component to component, and indirect component to component path is converted to a verification sequence (observer) even when they are subsumed by another path. For example, consider the threads in Fig. 6.1. The illustration partly encompasses three component-internal paths and two direct component to component paths. However, these are subsumed by the indirect component to component path (the end to end path) between the first and last thread. Applying complete coverage of each path type, even when they are subsumed, is often advantageous in practice if the architectural design contains faults. The locations where faults emerge become more evident if the satisfyabilities of subpaths of failed indirect paths also are explored. For example, given that the architectural design presented in Fig. 6.1 is free from erroneous behavior except for some behavior of the final thread, which entail an unsatisfiable indirect path, the satisfyability of its subpaths pertaining to the first and second thread would indicate that the erroneous behavior is apparent within the final thread. Moreover, the analysis of dynamic dependencies/independencies between the components requires inter-observer satisfyability checking with respect to the component-internal and direct component to component paths. The coverage criteria are therefore set according to the worst case load in the study.

6.2.2 Results

The results of the study are presented in Fig. 6.3- 6.14, derived from applications to a total of 45 different cases (architectural designs). The variation of cyclomatic complexity and thread and core quantities are distributed evenly over these cases, i.e., approximately 15 cases differ in the number of cores, 15 in the number of threads, and 15 in the cyclomatic complexity. The number of processor cores ranges from 1 to 25. The number of threads ranges from 1 to 100. And the cyclomatic complexity ranges from 1 to 1024. The average case corresponds to an architectural design with four processor cores, 25 threads,
Model checking

The model checking (execution of verification sequences) performance of AQAT with respect to outlier cases in terms of quantities of processors and threads is presented in Fig. 6.3. The tool is able to verify architectures with up to approximately 100 threads, given that the hardware platform of the architectural design is composed of a single-core processor and that the threads have a cyclomatic complexity of one. Such a verification process completes within 19.5 hours. In comparison, model checking of a single-core architecture with 10 threads is completed within 27 seconds. An application to an architecture with 110 threads under these premises results in an out of memory exception thrown by the UPPAAL model-checker. On the other hand, the tool is capable of verifying architectures with up to approximately 25 processor cores, given that a single thread with a cyclomatic complexity of one is allocated to each core. The verification process completes within 26.7 hours, where verification of architectures with additional processor cores results in an out of memory exception thrown by the model-checker. This does not correspond to the worst case scenario in terms of time consumption of the model checking process. An allocation of 25 independent threads to 25 processor cores yields only a total of 25 (component-internal) paths to verify. If the ratio of threads per core is increased by increasing the number of threads, and reducing the number of cores such that the model-checker may accommodate the combined complexity, the total time consumption increases due to a larger number of paths to verify. That is, until a certain threshold of the threads per core ratio is reached, since the other end of the spectrum corresponds to a single-core architecture with 100 threads, which is verified within 19.5 hours (less than 26.7).

The number of extracted paths with respect to the number of threads within the architecture is presented in Fig. 6.7. The most time-consuming model checking process of the studied samples corresponds to an architectural design with seven cores and 70 single-path threads, which completes within 161 hours. 189 paths (verification sequences) are extracted from such an architectural design, where each takes 3058 seconds to verify. Consequently, the worst case scenario in terms of time consumption per verified path is when the number of cores is maximized, as presented in Fig 6.8. The performance trend of an increasing number of cores is presented in Fig. 6.4. As indicated by the graphs,
6.2 Scalability

Figure 6.3

The time consumption of model checking increases exponentially with a growth rate of approximately 100% as the number of processor cores increases.

The model checking performance with respect to outlier cases in terms of cyclomatic complexity is presented in Fig. 6.5. Given an architecture with a single-core processor, a maximum cyclomatic complexity of 1024 (10 dual-path threads) can be verified. The process completes within 2.1 hours. Verification of architectures with higher cyclomatic complexities generates out of memory exceptions thrown by the verification sequences to observer automata transformation module. As presented in Fig. 6.7, the number of extracted verification sequences is approximately 2000 for a cyclomatic complexity of 1024. The time consumption per verification sequence is consequently four seconds (Fig. 6.8), which is significantly less compared to outlier cases in the dimensions of processor core and thread quantities: 3850 and 237 seconds respectively. The variance trend between a constant vs. an increasing cyclomatic complexity in architectural designs with increasing numbers of processors and threads is presented in Fig. 6.6. Note that the number of threads are divided equally among the cores. For example, an architecture of four cores and 20 threads corresponds to four independent pipelines, each of which composed of 5 threads. The cyclomatic complexity is consequently $2^5 \times 4$ and not $2^{20}$. Moreover, note that the rate of divergence is significantly higher with respect
to the complete model checking process. The number of extracted paths in the above example is 48 when the pipelines are composed of single-path threads and 280 when they are composed of dual-paths threads. The complete process is consequently completed within $48 \times 0.05\text{min} = 2.4\text{min}$ in the former case and $280 \times 3.26\text{min} = 886\text{min}$ in the latter.

The average model checking time consumption of all cases is 7.7 hours, where the average case corresponds to an architectural design with four processor cores, 25 threads, and a cyclomatic complexity of 51. A breadth-first search is generally the most efficient strategy in this study except when the cyclomatic complexity decreases close to one, where a depth-first search becomes the more efficient strategy.
6.2 Scalability

Model checking performance w.r.t. outlier cases:
Cyclomatic complexity

\[ y \approx 0.0003x^{1.2315} \]

Figure 6.5

Model checking performance trend: Cyclomatic complexity

Figure 6.6
Chapter 6. Evaluation

![Figure 6.7](image)

**Figure 6.7**

![Figure 6.8](image)

**Figure 6.8**
Framework Operations

The performance of framework operations with respect to single-core architectures with an increasing thread quantity is presented in Fig. 6.9. The most time-consuming operation is the generation of test cases based on analyses of trace files produced by the model-checker, which unexpectedly peaks at 80 threads (18 minutes) and decreases for larger thread quantities. The model transformation, verification sequences extraction, and verification sequences to observer modules generate a uniform increasing trend. Similar trends are generated in outlier cases of processor core quantities, as presented in Fig. 6.10. In the dimension of cyclomatic complexity presented in Fig 6.11, on the other hand, the verification sequences to observers transformation module dominates the time consumption in the higher end of the spectrum, which throws out of memory exceptions for higher cyclomatic complexities.

Figure 6.9
Many-core: time consumption of framework operations

Figure 6.10

Cyclomatic complexity: time consumption of framework operations

Figure 6.11
6.2 Scalability

Dynamic dependence analysis

The performance of dynamic dependence analysis is presented in Fig. 6.12. The data points are additionally presented on a logarithmic scale in Fig. 6.13. The time consumption of the analysis generally increases exponentially by the number of threads (and paths) in the architecture, where the growth rate increases by the number of processor cores as indicated by Fig. 6.12. Nevertheless, as indicated by Fig. 6.13, the time consumption of the analysis is higher and increasingly diverges for single-core platforms compared to multi-core platforms when the numbers of threads decreases.

Since the analysis does not induce any additional complexity to the timed automata model, it is scalable, in theory, according to the upper limits of the model checking process. However, complete dynamic dependence analyses are relatively time consuming for architectures with a high number of threads. For example, an analysis of a single-core architecture with 40 threads takes approximately 15 hours to complete. If the threads are allocated to a platform with four processor cores, the analysis completes within approximately 112 hours. By extrapolating these results, the time consumption of the analysis for outlier cases takes an unreasonable amount of time to complete, e.g. 39 days for a single-core architecture with hundred threads, without the support of a network of model-checkers. The scalability evaluation of dynamic dependence analysis is therefore delimited to a maximum of 40 threads in this study.

![Dynamic dependence analysis performance](image)

Figure 6.12
Chapter 6. Evaluation

Figure 6.13

Dynamic dependence analysis performance (logarithmic scale)

Figure 6.14

Number of possible dependencies
6.2 Scalability

6.2.3 Discussion

The results indicate that the resource consumption of framework operations is negligible compared to model checking. The framework operations together on average stand for 0.8% of the total time consumption of a project including model checking and test suite generation in this study. Optimization of the model checking process therefore has the largest potential effect on the improvement of scalability. A satisfiability checking parallelization of observers through distributed model-checkers increases the performance proportionately. Given that the observers may be distributed over an equal amount of model-checkers, the time consumption of the complete process will correspond to the time consumption per verification sequence presented in Fig. 6.8.

The time consumption per verification sequence is relatively low (four seconds) for an architecture with a high cyclomatic complexity (1024) in this study, as presented in Fig. 6.8. Although the complete model checking process takes 2.1 hours to complete when the cyclomatic complexity is 1024, the number of verification sequences to execute is extensive (approximately 2000). Nevertheless, the impact of an increasing cyclomatic complexity on the time consumption per verification sequence is significant when the number of cores and threads increases, as presented in Fig. 6.6. Consequently, an increasing cyclomatic complexity does not only increase the time consumption of model checking due to a larger number of verification sequences to execute, but also due to a significantly increased state space of the architectural design. Each parameter – cores, threads, and cyclomatic complexity – therefore significantly impacts the performance of the model checking process, regardless of the applied coverage criteria. An exponential regression analysis of the studied samples provides a rough estimate, with a determination coefficient of 77%, of the model checking process when a complete coverage criterion is applied:

\[ t \approx 2^{\text{cores}} \times 1,09^{\text{threads}} \times 1,006^{\text{CC}} \times 0,11 \] (model checking minutes)

The scalability of AQAT with respect to cyclomatic complexity is not limited by the capability of the utilized model-checker in this study, but due to the memory usage of the verification sequences to observer transformation module within AQAT. In fact, the results suggest that the model-checker is scalable to significantly higher cyclomatic complexities as the model checking time consumption per verification sequence is only 4 seconds when the cyclomatic complexity is 1024. The limitation may be improved by simple means, such as tuning of Java heap size. However, we are planning to implement a more suit-
able data management of the transformation process to achieve a sustainable solution.

An analysis of the unexpected trend of the test case generation module presented in Fig. 6.9 showed that the model-checker intermittently failed to produce trace-files (output satisfiability solution) when the complexity of the architectural design increases. This, in turn, causes the tool to produce incomplete test suites, with a portion of omitted test cases, which naturally takes less time to generate compared to a complete test suite. The frequency of omission tends to increase with an increasing complexity. However, this type of failure does not emerge for low to moderate architectural design complexities in this study. The root cause of these failures of the model-checker is currently being investigated but is to date unknown.

The results of the dynamic dependence analysis evaluation indicate that its performance highly depends on the structure of the architectural design. For a single-core architecture with a single pipeline of ten threads, which yields a total of 494 possible formulae to evaluate, the analysis completes within 87 minutes. For a dual-core architecture with two independent pipelines of five threads each, which yields a total of 414 possible formulae to evaluate, the analysis completes within 15 minutes. In both cases, the average time consumption per verification sequence of the model checking process is similar (one second), i.e., state space searches are performed equally fast. The time difference (87 vs. 15 minutes) is consequently caused by the differences in the proportion of dependences with respect to independences between the two cases, where the latter case is composed of a more independent structure compared to the former. The results consequently suggest that the order in which the formulae are evaluated significantly impacts the performance of the analysis, as theoretically predicted. In this study, we utilized a single strategy, which combines a smallest set first strategy with a largest set first strategy.
Chapter 7

Limitations And Future Work

The empirical evidence of utility is limited in the industrial study due to artificially created faults. Moreover, the frequencies of occurrence of the fault types are determined in the study by the frequency of occurrence of the corresponding syntactical elements in the architectural design. No studies have been conducted to investigate whether the approach generates a representative set of faults. An investigation of the representativeness of the injected faults with respect to empirical observations should therefore be conducted to reliably conclude the utility of the tool. Another approach is to evaluate the tool in real-time when applied to industrial practices with authentically created faults. Nevertheless, the fault injection module utilized in the study performs faulty changes to all types of architectural constructs that are transformed into timed automata and affect the behavior of the architectural design. A wide set of plausible real-life scenarios is therefore covered in the evaluation, but the occurrence frequencies and severities of the fault types in practice have not been studied.

By means of the scalability study, we discovered that AQAT inefficiently handles memory usage with respect to an increasing cyclomatic complexity of the architectural design under analysis. The solution is thereby not limited by the state space explosion problem when the cyclomatic complexity increases, as with quantities of processor cores and threads. Moreover, we discovered that the utilized model-checker intermittently failed to produce traces of state space searches when the complexity of the architectural design increases. This,
in turn, results in generations of incomplete test suites for complex architectural designs. The primary future work therefore includes improvements within these two areas. Finally, the results of dynamic dependence analyses suggest that the order in which the possible dependencies are evaluated significantly impacts resource consumption. A study wherein different strategies are evaluated to optimize the dynamic dependence analysis is therefore an interesting area for future work.

Regarding technical limitations of AQAF and AQAT, the developed algorithms for paths extraction do not consider potential control dependencies between branching (control) expressions, where paths that should not be able to execute by design nonetheless may be extracted as prescribed paths of the architecture. Such dependencies are complex to statically analyze, as the satisfiability of each control expression must be determined for each possible path and combination of inputs that lead to an execution of the expression. The user must consequently keep track of extractions of non-designed paths and ignore the corresponding verification verdicts in the current version of tool. A similar problem exists with conditional loops, where the paths extraction algorithms are unable to determine the potentially maximum number of iterations a loop should be able to execute by design. In the current version of the tool, the user is required to set upper bounds of loop iterations the tool should consider and keep track of extracted paths with loops that exceed the intentions of the design, such that the corresponding verification verdicts can be ignored. These limitations may be improved by symbolic execution, where infeasible paths are computed by SAT/SMT solvers. An integration of symbolic execution and SAT/SMT solvers with AQAF is consequently an interesting area for future work. Another area of improvement is compatibility with additional architecture description languages and runtime systems, which essentially is achieved by extending the transformation rules to timed automata and architecture flow graphs.

Regarding the performance of AQAT, the time consumption of state space searches for model checking and test suite generation may be reduced by distributing the computations to multiple computers. The UPPAAL environment is based on a client-server architecture, where model checking engines may be remotely installed on multiple servers and communicate with clients through TCP/IP. Future work includes a study wherein the effects of a model checking parallelization of AQAT are explored.

Slicing based on Architecture Dependence Graphs (ADGs) and inter-observer satisfiability independence analysis is primarily used in this thesis for impact analyses of architectural design changes. Nevertheless, the techniques
may also be used for hazard analysis and risk assessment, where potential fault propagations, common cause failures, and single point failures, etc., and estimations of their probabilities may be automatically calculated. Future work therefore includes studies wherein the usability of these techniques in the area of hazard analysis and risk assessment, such as automated generation of fault trees, FMEA tables, and HAZOP tables, is evaluated.
Chapter 8

Conclusion

In this thesis, the Architecture Quality Assurance Framework (AQAF) is presented and demonstrated as a holistic, systematic, rigorous, and automated solution to the verification of architectural engineering of embedded systems, from architectural requirements analysis and design to architectural implementation and maintenance. The development of AQAF is motivated by the increasing complexity of embedded systems, the cost and hazardousness of architectural faults, the limited system life-cycle coverage of the related work, and the limited industrial use of formal methods despite being considered as vital in the development of safety- and mission-critical embedded systems. AQAF provides a variety of formal verification techniques developed upon a joint formalism and semantic domain, architecture flow graphs (AFGs) and timed automata, to achieve a wide coverage of the system life-cycle and a fully automated and integrated solution. AQAF includes an architectural model checking technique to detect design faults, an architectural model-based test suite generation technique to detect implementation faults, and an architectural selective regression verification technique to efficiently detect maintenance faults.

The verification criteria of AQAF ensure correctness, completeness, and consistency of architectural control and data flows, which are extracted from architectural models by means of AFGs. In order to subject the models to model checking and model-based test suite generation, AQAF transforms the behavior of the architectural models into networks of timed automata. Verification sequences of the AFGs, in the form of control and data flow paths and their constraints, are subsequently transformed to observer automata that are
added to the timed automata model. Observers drive both the model checking process and the test suite generation process through reachability analysis. Satisfied observers indicate a complete, consistent, and correct architectural model whereas passed tests indicate conformance of the implementation with respect to the model. Finally, selective regression verification is performed by AQAF through slicing of architecture dependency graphs (ADGs) and dynamic dependence analysis by means of inter-observer satisfiability checking.

The Architecture Quality Assurance Tool (AQAT) implements the theory of AQAF and provides an easy adoption into industrial practices. An industrial safety-critical train control system is used to demonstrate its utility in practice and to evaluate fault-detection effectiveness and resource efficiency. Results indicate that the tool effectively detects design faults as well as implementations faults, with 100% and 98.5% rates of detection respectively, and that the selective regression verification technique efficiently detects faults introduced by maintenance modifications. The tool is additionally applied to a wide range of mock-up systems, including outlier cases, to evaluate the scalability of the tool and the capability of its embedded modules to handle a growing system complexity. The results suggest that the tool is scalable to rather complex systems, e.g. to systems with 100 software processes, 25 processor cores, or a cyclomatic complexity of 1024, without the support of distributed computing.
Bibliography


Bibliography

[49] 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.


[57] Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Compositional Verification of a Medical Device


