METHODS AND TOOL SUPPORT FOR ANALYZING ARCHITECTURAL MODELS OF EMBEDDED SYSTEMS

Stefan Björnander

2012
Abstract

Embedded systems are ubiquitous in the modern world. They are microcomputers most often included in complete devices consisting of software and hardware. Embedded systems range from small devices to large systems monitoring and controlling complex processes. The design and development of such systems is a complex task, since embedded systems often need to fulfill extra-functional requirements, on top of functional ones, within constrained amounts of platform resources. Some embedded systems are safety-critical, meaning that some system failures may cause severe damage in human life or large financial losses.

One way to ensure that a system works in accordance with its specification is to define the system in an Architecture Description Language (ADL) and apply formal verification methods on the architectural model, well in advance of the system implementation. The Architecture Analysis and Design Language (AADL) of the Society of Automotive Engineers (SAE) has become popular in, e.g., the avionic and automobile industry. The AADL specification holds several annexes, including the Behavior Annex that is considered in this thesis. Parts of AADL lack a formal semantics, which prevents formal analysis of AADL models, e.g., model checking. Moreover, AADL does not support time annotations, which makes modeling of real-time systems harder.

In this thesis, we address the above-mentioned shortcomings by presenting a formal analysis framework including a denotational semantics for a subset of AADL and its Behavior Annex, which allows for checking properties defined in Computation Tree Logic (CTL) by model checking. Model checking is a formal verification method that has proved to be powerful as well as effective in uncovering design errors prior to system implementation. Our AADL-semantics is supported by a model-checker, called ABV, which implements the semantics in Standard ML and is encapsulated in an Eclipse plug-in. We also present a time annotation extension of AADL, implemented in a tool that translates the latter and its Behavior Annex into the Timed Abstract State Machine (TASM) language for simulation of real-time features.

Effective component distribution, in particular achieving optimal component distribution, against a set of constraints (e.g., frequency and bandwidth) is a problem tightly connected to architectural modeling. In order to address this problem, we have developed a tool that performs near-optimal component distribution with respect to a series of parameters.

The research results, which have been illustrated on relevant case studies, provide to the system engineer means for proving that the architectural system model meets its specification with respect to both function and timing. The research has been conducted in the context of the PROGRESS research center, for predictable embedded software systems.
I dedicate this thesis to my parents Ralf and Gunilla, my sister Catharina, her husband Magnus, and their sons Emil and Rasmus.
Acknowledgments

I would like to thank my advisors Paul Pettersson, Kristina Lundqvist, and Cristina Seceleanu at Mälardalen University, Lars Grunske at Swinburne University, and Rikard Land at CrossControl AB, as well as Patrick Graydon and Antonio Cicchetti at Mälardalen University.

This research work was partially supported by Swedish Research Council (VR), and the Swedish Foundation for Strategic Research via the strategic research center PROGRESS, ARTEMIS JU, Vinnova, and CrossControl AB in the SafeCer project (JU Grant Agreement number 269265).

Stefan Björnander
Västerås, December, 2012
List of Publications

Publications Included in the Licentiate Thesis


Other publications, not included in the thesis


# Contents

## I Thesis

1 Introduction  
1.1 Preliminaries .................................................. 5  
1.1.1 Embedded Systems ........................................ 5  
1.1.2 Model-Based Engineering .................................. 5  
1.1.3 Software Architecture ...................................... 6  
1.1.4 Architecture Description Languages ....................... 6  
1.1.5 AADL ......................................................... 6  
1.1.6 Formal Verification Techniques: Model-checking ........... 8  
1.2 Thesis Overview ................................................. 10  

2 Research Problem .................................................. 13  
2.1 Problem Description ............................................. 13  
2.2 Research Questions ............................................... 13  

3 Research Results .................................................. 15  
3.1 A Denotational Semantics for AADL and its Behavior Annex .... 15  
3.2 ABV – A Verifier for the AADL and its Behavior Annex .......... 16  
3.3 AADLtoTASM ...................................................... 16  
3.4 ArcheOpterix ...................................................... 17  
3.5 Questions Revisited ............................................... 17  

4 Research Method .................................................... 19  

5 Related Work ....................................................... 21  
5.1 Formal Semantics of AADL ........................................ 21  
5.2 Tools for Model Checking AADL ................................ 21  
5.3 Architecture Deployment ......................................... 22  

6 Conclusions and Future Work ....................................... 23  
6.1 Contributions ..................................................... 23  
6.2 Future Research Directions ....................................... 24  

Bibliography .......................................................... 25  

## II Included Papers

7 Paper A:  
A Formal Analysis Framework for AADL  
7.1 Introduction ..................................................... 33  
7.2 Preliminaries ..................................................... 33  
7.3 The Semantics of AADL Structural Elements ..................... 35
Paper B:
ABV – A Verifier for the Architecture Analysis and Description Language (AADL)
8.1 Introduction ......................................................... 47
8.2 ABV Tool Overview ............................................... 48
8.3 Tool Input .......................................................... 48
8.3.1 The AADL and its Behavior Annex Model ..................... 49
8.3.2 The CTL Property Specification .................................. 50
8.4 The Graphical Interface ............................................. 51
8.5 Case Study .......................................................... 52
8.6 Related Work ....................................................... 53
8.7 Conclusions and Future Work ...................................... 54
Bibliography .......................................................... 57

9 Paper C:
Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract
State Machines ........................................................ 59
9.1 Introduction ........................................................ 61
9.2 Background ......................................................... 61
9.2.1 The Architecture Analysis and Description Language ............ 61
9.2.2 The Timed Abstract State Machines .............................. 62
9.3 Time and Resource Extensions of the AADL Behavior Annex ......... 63
9.3.1 The AADL Behavior Annex Extension ............................ 63
9.3.2 Translation Semantics ............................................. 63
9.3.3 Tool Support ...................................................... 64
9.4 Case Study .......................................................... 65
9.5 Related Work ....................................................... 69
9.6 Conclusions and Future Work ...................................... 70
Bibliography .......................................................... 71

10 Paper D:
ArcheOpterix: An Extendable Tool for Architecture Optimization of AADL Models
10.1 Introduction ........................................................ 75
10.2 Architecture and Tool Description .................................... 76
10.2.1 Overview of the Tool .......................................... 76
10.2.2 The OSATE Interface ........................................... 77
10.2.3 Architecture Analysis Module (AAM) ............................ 78
10.2.4 Architecture Optimization Module ............................... 79
10.2.5 AADL Model Generator ......................................... 80
10.3 First Experimental Results ......................................... 81
10.4 Related Work ....................................................... 82
10.5 Conclusions ........................................................ 84
Bibliography .......................................................... 86
I

Thesis
Chapter 1

Introduction

An embedded system is a system made of software and hardware, designed for a set of specific tasks [1]. Embedded systems play a vital role in, e.g., aerospace applications, automotive systems, communication systems, air traffic control, railway signaling, and medical equipment. The design and development of such systems is challenging because the specification of embedded systems often should be met within limited platform resources (e.g., available memory). In addition, they are often required to meet extra-functional requirements, such as constraints in timing or physical space. An airbag is an example of an embedded system with timing constraints: it should not only function in accordance to its specification, it should do so within a certain time frame. In case of vehicle collision the airbag should not inflate too early, no too late. A smart phone is an example of an embedded system that performs complex functioning within limited resources. We describe embedded systems in more detail in Section 1.1.1.

Ensuring system correctness is particularly important for the design of safety-critical embedded systems [2]. A safety-critical system is a system where failures may cause serious damage to people or valuable assets. An example is the brake system of a car. If it fails, it may (indirectly) seriously hurt the passengers and other road-users. The term correctness may be interpreted in several ways. Projects may have specifications that are not easily checked by a formal reasoning, e.g., a user-friendly interface. Therefore, in this thesis we checks the kind of correctness that is suitable to define in a formal state machine.

Given the stringent requirements of embedded systems, it would be beneficial to ensure already at the early development stages that the system meets its specification, since it tends to be more difficult to rectify mistakes in the later phases. If an error necessitates an architectural redesign, it would be much more difficult to tackle such task during later phases of the implementation. Of high practical interest is the architecture design phase; that is, uncovering as early as possible architectural mistakes that would impede not only the system’s correct functioning, but also the system’s resource usage, performance, maintainability and other extra-functional aspects. Consequently, the development process for embedded systems should include verification techniques in the architecture design phase to provide evidence that a system’s architecture has the potential to fulfill its functional as well as extra-functional requirements.

The Architecture Analysis and Design Language (AADL) is a powerful language for describing models of the system’s architecture, which has become popular in the avionic and automobile industry [3]. It is more closely described in Section 1.1.5. Parts of AADL still lack formal semantics, which limits the possibility to perform formal analysis of AADL models. Moreover, the latter are not executable, which limits the possibility to perform automated property checking. Consequently, it is highly desirable to try to overcome the mentioned limitations of AADL. To do so, AADL needs to be formally defined, as any attempt to achieve formal verification requires a precise mathematical model. It would also be beneficial that the analysis techniques based on the semantics are supported by tools integrated into an AADL tool chain, which would make it easier for a user with limited knowledge of the underlying formalism to perform, e.g., model checking of AADL models.

The goal is even more justified as AADL is also equipped with a number of annexes, out of which the Behavior Annex (that is in our current focus) models the AADL behavior as abstract state machines. We describe the Behavior Annex in Section 1.1.5.

We address the above-mentioned limitations of AADL by providing a formal analysis framework of
its models, which includes a denotational semantics for a subset of AADL and its Behavior Annex. The framework facilitates model checking of properties defined in Computation Tree Logic (CTL), which is a branching-time temporal logic more closely described in Section 1.1.6. The framework includes the semantics implemented in the functional programming language Standard ML. It also holds the ABV model checker [4], which encapsulates the semantic implementation and provides a graphical user interface based on the Eclipse Framework. ABV includes a parser that translates AADL source code (which is defined by the AADL syntax) into Standard ML in a format suitable as input to the semantics implementation. With the help of ABV, the user is able to verify a subset of CTL properties of the model without knowledge of the underlying technical AADL notation.

A way to provide timing reasoning regarding the architecture is to add time annotations to the architecture language. Consequently, we have added time annotations to AADL and its Behavior Annex and implemented a translation tool that generates code for the Timed State Machine Language (TASM) [5], which makes it possible to simulate real-time behavior. It is also possible to further translate the TASM code into UPPAAL [6], which provides the possibility to formally reason about real-time features.

Another problem regarding architecture modeling concerns component distribution. When trying to optimize the component deployment on a particular platform, it might happen that several conflicting demands (such as inter-component interaction frequency or reliability and delay of links between hosts and components) need to be satisfied. In addition, there may be several other constraints (such as available memory and constraints in component connections) to consider. One way to solve these problems is to use an evolutionary algorithm; that is, a generic population-based optimization algorithm [7, 8]. The evolutionary algorithm uses some mechanisms from biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem correspond to the role of individuals in a population, and a fitness function determines the environment within which the solutions are regarded as live. Evolution of the population then iterates over application of the operators. In this thesis, we present such a solution to the optimization problem, within the ArcheOpterix tool, which distributes components on a system in a way optimal with respect to network speed, available memory, restraint in communication, localization between hosts and components.

When chosen our AADL subset, we have tried to select a generic kernel that would make it possible to define formal specifications for general systems. Therefore, we have limited the subset to the generic system component. There are two types of systems: (i) the system that defines the port interface and an optional behavioral annex, and (ii) the system implementation that defines the subcomponents and the port connections between them. The subcomponents of the system implementation are instances of earlier defined systems (equivalent to objects and classes in object-oriented languages) and the connections are made between input and output ports of the subcomponents, rather than the systems. The subset does also include the Behavior Annex with (i) states (among which one is the initial state) and state variables (which can be initialized); (ii) input and output signals that are connected to the ports of the surrounding system (together with initializations of the output signals), and (iii) the set of state transitions [9] where each transition is equipped with a guard; that is, a Boolean expression that has to evaluate to true for the transition to be fired.

In brief, the core of our contributions lies in the area of modeling and analyzing architectural models, and consists of the following:

- A formal analysis framework including a denotational semantics of a subset of AADL and its Behavior Annex, with an implementation in Standard ML encapsulated in an Eclipse plug-in.

- A time annotation extension of AADL, implemented in a tool translating time annotated AADL and its Behavior Annex into TASM for simulation of real-time features.

- A tool that performs near-optimal component distribution with respect to a series of parameters.

---

1 The Eclipse Framework is described at http://www.eclipse.org.
1.1 Preliminaries

In this section, we review the background needed for the remainder of the thesis. We recall the basics of embedded systems, software architectures, model-based engineering, AADL, and formal verification.

1.1.1 Embedded Systems

Physically, an embedded system may range from a portable device, such as a GPS-receiver, to a large system, such as a nuclear power plant controlling system. In the same way, the system complexity may range from a single microcontroller chip to a high level of complexity involving systems with multiple units and peripheral devices.

An embedded system is designed to perform an increasingly large set of dedicated tasks, often within constrained resources. It is controlled by a main processor core, typically a microcontroller [1], and it often consists of small computerized interconnected components.

There are many kinds of embedded systems. Telecommunication systems offer a wide range of embedded systems, from telephone switches to end-user mobile phones. Another field where embedded systems are common is in transportation, e.g., flight or automobile systems. A modern car is equipped with up to eighty embedded processors that control, for instance, the anti-lock braking systems or the automated four-wheel drive.

Some embedded systems also provide elements of programming abilities. For instance, a handheld computer may be viewed as an embedded system – it has microcomputers and operating systems, often with its own battery power. However, it is not a truly embedded system as it allows different applications to be loaded.

Real-Time Systems. A real-time system is an embedded system that in addition to functional and extra-functional requirements has to meet timing requirements (e.g., meeting a deadline [10, 11]). There are two kinds of deadlines: hard and soft. A hard deadline is one that has to be met under any circumstances, or otherwise the system fails with possibly catastrophic consequences. For instance, an airplane ejection seat needs to work within a certain time frame, otherwise the pilot would not be saved in case of a crash. A soft deadline is one that should be met most of the time; that is, sometimes a bounded tardiness is accepted. For instance, a system that maintains and updates information of commercial airlines can be viewed as a real-time system with soft deadlines. It is desired that the information be displayed within a certain time frame, while the system is still operational with some latency.

Safety-Critical Systems. A safety-critical system is a system that is not allowed to fail [12]. Knight states that “safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment” [13]. Examples of such systems are the control systems of airplanes, space shuttles, and nuclear power plants. An example of a safety-critical system that failed is the French Ariane rocket that in 1996 was destroyed shortly after launch, due to a flaw in the program code of the control software. Given the potential failure consequences, formal analysis of safety-critical systems prior to implementation is highly recommended.

1.1.2 Model-Based Engineering

This thesis relies on the Model-Based Engineering (MBE) paradigm. MBE is a software development methodology focusing on modeling domain information and requirements rather than algorithmic concepts [14]. The MBE approach is meant to increase productivity by encouraging reuse of standardized models and to simplify the process of design with reuse of design patterns as well as promoting communication between individuals via standardized terminology [15].

MBE refers to a range of development approaches based on the use of software modeling as its primary form of expression. Some models are constructed to a certain level of detail, on which source code needs to be written in a separate step, while other models are detailed to the extent that source code can be generated automatically from the models, ranging from system skeletons to completed products.
The first tools to support MBE were the Computer-Aided Software Engineering (CASE) tools developed in the 1980s. MBE has become very popular since the introduction of the Unified Modeling Language (UML), designed by Grady Booch [16], Jim Rumbaugh [17], and Ivar Jacobson [18].

However, one common problem with MBE is that the model becomes changed during the system lifetime without the change being mirrored in the code, or vice versa. To address that problem, tools were developed to be able to support traceability. These tools identify discrepancies between the model and code, generate code from the model, or vice versa. One of the most wide-spread tools to support MBE is the Eclipse Modeling Framework (EMF) [19].

1.1.3 Software Architecture

As our main focus is the analysis of embedded system architectural models we introduce the notion of software architecture in the following, and briefly overview the languages that allow its description.

The software architecture of a system is the set of structures needed to reason about the system. The system comprises related software and hardware elements. The discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. However, there is still no agreement on the precise definition of the term.

The origin of the field can be traced back to Edger Dijkstra [20] and David Parnas [21]. They emphasized the structure of a software system and claimed that getting the structure right is critical. Mary Shaw and David Garlan wrote the book *Software Architecture: Perspectives on an Emerging Discipline* [22], which introduced concepts in Software Architecture, such as components and connectors. The IEEE standard *Recommended Practice for Architecture Description of Software-Intensive Systems* [23] was the first formal standard in the area of software architecture. The need to formally reason about system architecture eventually resulted in the development of architecture analysis and design languages, described in the next section.

1.1.4 Architecture Description Languages

An Architecture Description Language (ADL) is a computer language used to describe and represent software architectures. An ADL is suitable for communicating an architecture to all interested parties, while bearing a formally defined syntax and semantics. Usually, ADLs support the tasks of architecture creation, refinement and validation, and are intended to be both human and machine readable, and provide a basis for further implementation [24]. However, ADLs may differ in their ability to handle real-time constraints at the architectural level and support the specification of different architectural styles.

Several ADLs have been developed, such as Acme (developed by CMU) [25], EAST-ADL [26], Darwin (developed by Imperial College London) [27], Wright (developed by CMU) [28], and AADL (standardized by SAE, described in the next section) [29], to mention a few.

1.1.5 AADL

AADL [29, 30, 31] is an expressive language intended for the design of both the hardware and the software of a system. AADL is a standard of the Society of Automotive Engineers (SAE) and is based on MetaH and UML [32]. AADL is constrained in one respect: it addresses the architecture stage of the system’s life cycle (software components mapped on hardware components), but the earlier stages cannot be addressed within the language [33]. AADL uses component abstractions at different levels, which can be grouped in three categories, as follows:

1. Application Software.
   - Thread. Threads can execute concurrently and be organized into thread groups
   - Thread Group. Thread groups are component abstraction for logically organizing threads or thread groups components within a process.

---

2 AADL is described at http://aadl.info.
3 SAE is presented at http://www.sae.org.
• **Process.** A process is a protected address space whose boundaries are enforced at runtime.

• **Data.** Data refers to data types and static data.

• **Subprogram.** A subprogram is a model of a subprogram component that represents a callable piece of source code.

2. **Hardware (Execution Platform).**

• **Processor.** A processor schedules and executes threads.

• **Memory.** Memory stores code and data.

• **Device.** A device represents sensors and actuators that interface with the external environment.

• **Bus.** A bus interconnects processors, memory, and devices.

3. **System Component.**

System components are composites that can consist of other system components, as well as software or hardware components. The component types are defined using a parameterized set of properties. Furthermore, components communicate with each other through ports. It is possible to define physical port-to-port connections, as well as logical flows through chains of ports. Component definitions are divided into component types holding the public (visible to other components) features, and component implementations that define the private parts of the component. The AADL standard includes runtime semantics for mechanisms of exchange and control of data, including message passing, event passing, synchronized access to shared components, thread scheduling protocols, and timing requirements.

AADL can be used to model and analyze systems already in use, as well as to design new systems. It can also be used in the analysis of partially defined architectural patterns. Moreover, AADL supports the early prediction and analysis of critical system qualities, such as performance, schedulability, and reliability. Within the core language, property sets that add new properties for components can be declared. Additional models and properties can also be included by utilizing the extension capabilities of the language [3].

AADL components interact through defined interfaces. A component interface consists of directional flow through data ports for state data, event data ports for message data, event ports for asynchronous events, subprogram calls, and explicit access to data components. Application components have properties that specify timing requirements such as period, worst-case execution time, deadlines, space requirements, and arrival rates [34].

There are a number of AADL tools developed that provide automated support for system design and analysis. One of them is the Open Source AADL Tool Environment (OSATE4), which is a plug-in for the Eclipse Development Environment. OSATE supports analysis and simulation of AADL models.

In order to increase the expressiveness of AADL, it is possible to add annexes to the pure architectural descriptions. One of them is the Behavior Annex [35] that models an abstract state machine [36]. Each component’s function and logic can be described by a corresponding behavior model, which consists of three parts [9]:

• **States.** The machine states, one of them is the initial state.

• **Transitions.** The condition for a transition from one state to another (or between the same state) is determined by a guard: an expression evaluated to a logical value.

• **State Variables.** The state variables are similar to variables in programming languages; they can be inspected and assigned.

Listings 1, 2, and 3 hold an example of a system modeled in AADL. This system, originally introduced by Björnander et al. [37], is illustrated in Figure 1.1. There is one main system and two subcomponents. Each subcomponent has a behavior annex with a critical state. The subcomponents hold a waiting and critical state each, and communicate by ports. Each component has a behavior annex that makes sure the components never hold the critical state at the same time, which can be proven with CTL (See Section 1.1.6).

---

4Available at http://www.aadl.info/aadl/currentsite/tool/osate.html.
1.1.6 Formal Verification Techniques: Model-checking

Formal verification is the process of rigorously proving the correctness of a system design expressed as an abstract mathematical model. That is, that the design meets the system requirements. In this thesis, the result of the process is a “yes/no” answer. One of the well-known formal verification techniques is model checking. Model checking has gained popularity in industry, where it has been used for verifying software and hardware designs, e.g., safety-critical code in aerospace industry (Boeing) [38].

Model checking is one of the most popular formal analysis methods. It focuses on automation, and the challenge is to develop algorithms and software implementing such algorithms, to perform model checking in an efficient way. An automated model-checking tool accepts a model and a formalized property, and decides whether the property is true with respect to the model. By traversing the system state space in an exhaustive manner.

There are three types of properties that we are interested in: safety (something bad never happens), liveness (something good will always eventually happen), and reachability (on some path, something good will eventually happen). In case of a safety property that is violated, the model-checker has the obligation to report a “counter-example,” that is, a trace from the initial state that ends in a state in which the property does not hold. In case of a reachability property, the tool reports a trace from the initial system state to a state satisfying the property.

Formally, the model-checking problem can be stated as follows: given a desired property, expressed as a temporal logic formula \( p \) (in our case CTL), and a model \( M \) with initial state \( s \), decide if \( M, s \models p \). Clarke and Wing [39] give an extensive overview of model checking.
Listing 1 The Main System.

system implementation MainSystem.impl
subcomponents
  subsystem1 : system Subsystem1;
  subsystem2 : system Subsystem2;
connections
  event port subsystem1.CriticalLeave -> subsystem2.CriticalEnter;
  event port subsystem2.CriticalLeave -> subsystem1.CriticalEnter;
end MainSystem.impl;

Listing 2 Subsystem 1.

system Subsystem1
features
  CriticalEnter : in event port;
  CriticalLeave : out event port;
annex SubsystemAnnex1
{**
  initializations
    CriticalLeave !;
  states
    Waiting : initial state;
    Critical : state;
  transitions
    Waiting ![CriticalEnter?] -> Critical :
    Critical ![true] -> Waiting
    {CriticalLeave !;}
**};
end Subsystem1;

Computation Tree Logic. CTL is a branching-time temporal logic, in which one can express the system properties that are going to be verified. The state space of a system is considered a tree structure with a non-determined future. There are different paths into the future and any one of them may be the one realized. For a thorough description of CTL we refer the reader to the work of Huth and Ryan [40]. In the following, we describe a set of derived CTL operators. The CTL operators are of two kinds: quantifiers over paths, All ($A$) and Exists ($E$); and path-specific temporal operators, Global ($G$) and Eventually ($F$). See Table 1.1 and Figure 1.2 for a closer description.

When generating CTL formulae, the following partial grammar can be applied:

1. In Table 1.1, $A\phi$ and $E\phi$ are quantifiers, and $\phi$ is a path formula.

2. $\phi$ is a path formula of the form: $G\psi$ and $F\psi$, where $\psi$ is a state formula.

By combining (1) and (2) one can generate the language of CTL. For instance, a property of the form $AG\phi$ is called a safety property, as it has to be true in all states of all paths of the execution. Another example would be the reachability property described in CTL as $EF\psi$; it means that there exists a path where eventually $\psi$ evaluates to true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A\phi$</td>
<td>$\phi$ must be satisfied for every child.</td>
</tr>
<tr>
<td>$E\phi$</td>
<td>$\phi$ must be satisfied for at least one child.</td>
</tr>
<tr>
<td>$G\psi$</td>
<td>$\psi$ must be satisfied for each node on the path.</td>
</tr>
<tr>
<td>$F\psi$</td>
<td>$\psi$ must be satisfied for at least one node on the path.</td>
</tr>
</tbody>
</table>

Table 1.1: A set of operators derived from a subset of CTL.

In Listing 4, a CTL safety property specification has been formulated, operating on the system described in Section 1.1.5. The $A$ and $G$ operator combination decides whether the expression is always evaluated to true, meaning that the two subsystems never reach their critical sections at the same time.
Listing 3 Subsystem 2.

```plaintext
system Subsystem2
features
  CriticalEnter : in event port;
  CriticalLeave : out event port;
annex SubsystemAnnex2
{**
  states
    Waiting : initial state;
    Critical : state;
  transitions
    Waiting −[CriticalEnter?]−> Critical;
    Critical −[true]−> Waiting
    {CriticalLeave !;}
}**
end Subsystem2;
```

Figure 1.2: CTL Operator Combinations.

1.2 Thesis Overview

The remainder of the thesis is divided into two distinct parts. The first part is a summary of the performed research. Chapter 2 formulates the main research goal and introduces the research questions that support the goal. Chapter 3 describes the research results and recapitulates the research questions, in the light of the described results. The steps we have followed in our research method are described in Chapter 4. We survey related work in Chapter 5, whereas in Chapter 6 we conclude the thesis, and summarize the contributions and outline future work that formulates guidelines for further graduate studies.

The second part of the thesis presents a collection of conference and workshop papers that contain the main research results that address our research questions. The following four papers are included in the second part of the thesis:


Summary: This paper describes a denotational semantics for a subset of AADL and its Behavior Annex. The semantics includes the possibility of defining properties specified as Computation Tree Logic (CTL) formulae. Our contribution sets the premises for the formal verification of AADL models, enhanced with behavior.

Contribution: I was the main driver of the paper. I defined the AADL semantics and wrote most of the paper. My coauthors have contributed ideas, discussions, and reviews.


Summary: The AADL with Behavior Annex Verifier (ABV) is a tool for model checking real-time systems defined in the language. It performs model checking by constructing a state tree that is later traversed with respect to a question, expressed in the Computation Tree Logic (CTL). The tool implements the semantics introduced in paper A in Standard ML and encapsulates the semantics in an Eclipse plug-in. The
Listing 4 CTL Property Specification.

\(\forall \text{global} \ \neg (\text{subsystem1}. \text{Critical} \ \text{and} \ \text{subsystem2}. \text{Critical})\)

tool is exercised on the Production Cell case study.

**Contribution:** This paper was mainly written by me. I implemented the semantics in Standard ML and encapsulated it in an Eclipse plug-in. My coauthors contributed ideas, discussions, and reviews.


**Summary:** This paper presents an extension of the behavior model of AADL using time annotations in order to improve the evaluation of timing properties in AADL. The translational semantics of this extension is based on mappings to the Timed Abstract State Machines (TASM) language. As a result, timing analysis with timed simulation or timed model checking is possible. The translation is supported by an Eclipse-based plug-in and the approach is validated with a case study of an industrial production cell system.

**Contribution:** I was the main writer of the paper and I also implemented the tool. My coauthors contributed ideas, discussions, and reviews.


**Summary:** For embedded systems, quality requirements are equally important than functional requirements. The foundation for fulfilling these quality requirements has to be set in the architecture design phase. However, finding a suitable architecture design is a difficult task for software and system architects. The reasons for this include the ever-increasing complexity of todays systems, strict design constraints and conflicting quality requirements. To simplify the task, this paper presents an extendable Eclipse-based tool called ArcheOpterix. The tool provides a framework to implement evaluation techniques and optimization heuristics for AADL specifications. Experiments with a set of initial deployment architectures provide evidence that ArcheOpterix can successfully find solution architectures with better quality.

**Contribution:** This paper was written with equal contribution of all the authors. I contributed mainly by implementing the architecture deployment tool.
Chapter 2

Research Problem

This chapter presents the scope of our work by formulating the research goal, and introducing the research questions that address the goal.

2.1 Problem Description

Architectural models are of outmost importance for the quality of the resulting system, hence our research focuses on methods for formal verification of software architectures. In particular, in this thesis, we address the problem of formal verification of AADL models, by model checking, in an attempt to improve the capability of detecting possible architectural bugs already at early-stages of embedded system design. The overall goal of the research is:


To support this goal, in Section 2.2, we formulate four research questions that we address in our work.

2.2 Research Questions

When constructing safety-critical embedded systems, it is essential that they perform the tasks in accordance with their specifications. In order to reach this goal, it is beneficial to model the system architecture in an architecture analysis language, such as AADL. It is also advisable that the language defines the architecture in a way that is formally unambiguous as well as easy to implement. These language properties would then set the premises for carrying out formal verification of the model, hence for potentially uncovering trouble spots at an early design-stage. The first research question that addresses this issue is:

Q1: How to formally describe AADL in an implementable way that is amenable to formal analysis?

When the language has been formally defined, one has the premises to verify that the system is correct with respect to certain requirements. There are several methods to do so, out of which model checking and simulation are in our focus. Therefore, the second research question is:

Q2: How to support the analysis of AADL descriptions, in order to guarantee that an embedded system architecture satisfies given functional requirements?

When dealing with real-time systems, timeliness is essential. When modeling a system, it is then advisable to include time annotations to the component and subsystems described in the architecture analysis language, such as AADL, that is, time annotations that can be analyzed or simulated. The third research question is:
Q3: *How to model timed behavior of AADL components and subsystems?*

For embedded systems, extra-functional requirements (such as timing, reliability etc.) are equally, or sometimes even more important than functional requirements. One particular problem is to find a component deployment that is near optimal, with respect to certain constraints, such as frequency and message size of interaction between components and reliability and network delay of network links between hosts of components. To tackle this problem, we try to answer the fourth research question, that is:

Q4: *How to provide support for near-optimal component deployment within given constraints?*

In Section 3.5 we discuss how Questions 1 to 4 are addressed in Papers A to D.
Chapter 3

Research Results

This chapter presents our contributions and research results, starting from the research questions proposed in Chapter 2. The research partially provides answers, and gives directions for further research. The following sections describe each research contribution, respectively.

3.1 A Denotational Semantics for AADL and its Behavior Annex

**Goal:** The goal of this research is to formally define a denotational semantics for AADL and its Behavior Annex, in order to describe such models unambiguously.

AADL is an internationally used industry standard language for modeling embedded systems. However, it lacks a formal semantics of the elements involved in building architectural and behavioral models. When designing safety-critical embedded systems, the lack of formal semantics is especially important to address because failures may have serious consequences. Moreover, AADL models are not executable, which limits the possibility to analyze their safety and liveness properties. Consequently, it is highly desirable to overcome these limitations of AADL. To achieve the latter, one needs to equip AADL with formal semantics, in order to open the possibility of formally verifying such models. Most desirably, the respective formal semantics should be supported by tools integrated into an AADL tool chain. In order for the semantics to be easily verifiable, it is also desirable that the gap between the semantics and some programming language is as small as possible.

Denotational semantics [41] is an approach to formalize the meaning of programming languages by constructing mathematical objects (called denotations), which describe the meaning of expressions from the languages. The main benefit of denotational semantics is that it is based on a rigorous mathematical foundation, and is built on the same principles as functional programming languages, such as Standard ML [42]. In brief, our goal is to give AADL and the behavior Annex a precise semantics that is easily translatable into a programming language.

**Research Process:** We soon realized that it would become an overwhelming task to define a rigorous semantics for the complete AADL and its Behavior Annex. Therefore, our first task was to identify a generic subset of AADL and its Behavior Annex that we were to define formally. We decided to include the system, system implementation, features, subcomponents, and connection parts of AADL, as well as the whole Behavior Annex. Once chosen, we described in a denotational semantics style the respective elements.

**Results:** The main result of this part of research is the denotational semantics, which defines the subset of AADL and its Behavior Annex mentioned above, formally and unambiguously [37, 43].

**Limitations and Future Work:** As the denotational semantics that we have proposed does not cover the whole of AADL, but rather a subset, the next task would be to extend the semantics to all the elements of the language.
3.2 ABV – A Verifier for the AADL and its Behavior Annex

**Goal:** The goal of this research is to implement a tool that performs model checking of AADL and its Behavior Annex descriptions. In order to guarantee that a system’s architecture meets its specified requirements, an automated verification tool tailored to AADL and its Behavior Annex is an essential ingredient.

**Research Process:** As mentioned in Section 3.1, one major benefit of the denotational semantics is that it is based on the same principles as functional languages. Therefore, the first step was to review the most popular functional languages and chose one for our implementation purposes, based on the semantic gap to the denotational semantic characterization of AADL elements. We have identified Standard ML as being a suitable option, semantically, to our description of AADL and its Behavior Annex. Hence, we implemented the denotational semantics of the chosen elements in Standard ML, and encapsulated the tool around the implementation.

**Results:** The results of this research are the ABV system architecture verifier [4]. It performs model checking of AADL models of embedded systems. Its input is an AADL model source code file and a CTL requirement specification. ABV is an Eclipse plug-in, providing seamless interaction with the Eclipse environment. Moreover, ABV encapsulates a Standard ML implementation of the formal denotational semantics that unambiguously defines a subset of AADL and its Behavior Annex.

**Limitations and Future Work:** Since the tool is based on the denotational semantics of Section 3.1, it suffers from the same limitations concerning the AADL subset. The evaluation of the CTL properties is also somewhat cumbersome: a complete state space is generated and evaluated. It should be possible to perform the evaluation “on-the-fly,” i.e., to evaluate a CTL property while generating the state-space and possibly to avoid generating the whole state space. Another limitation is that we cannot verify nested CTL properties. We intend to try to remove these limitations in our future work.

3.3 AADLtoTASM

**Goal:** The timing behavior and resource consumption of systems depend heavily on the architecture chosen for these systems. Furthermore, architectural mistakes that cause a system not to fulfill certain real-time requirements are hard to correct in later development phases. Consequently, the real-time requirements of a proposed architecture should be checked as soon as that architectural solution is available. For this to be possible, we need to be able to express timing behavior and real-time properties of AADL models. Hence, the goal of this research is to develop a tool that analyzes real-time properties of AADL models for embedded systems. We have chosen TASM, which is a language based on the timed automata, with some extensions.

**Research Process:** First, we inspected the syntax of the Behavior Annex and added time annotations that made it possible to annotate each transition with a fixed time or a time interval. Then we matched each feature of the Behavior Annex to a corresponding feature of TASM, and implemented a translator from AADL to TASM [44].

**Results:** We have developed a translational semantics that maps the extended behavior specifications of AADL into a network of timed abstract state machines. As a result, existing evaluation and verification techniques, such as timed simulations and timed model checking defined for the TASM specification formalism can also be applied to extended AADL specifications. The translation of time-extended AADL specifications into TASM is supported by a tool called AADLtoTASM. The overall approach is validated with a case study of an industrial production cell.

**Limitations and Future Work:** We are aware that, for this work also, there are some issues that can be addressed in the future. For instance, the stopping criterion for the simulation can be improved; that is, a special stop-state is added to the model that stops the simulation. One possible extension of this
work is to further translate the TASM model into UPPAAL, in order to perform timed model checking. If accomplished, it would mean that it would be possible to, in detail, formally define best-case and worst-case time behavior of the AADL model.

3.4 ArcheOpterix

Goal: We have already argued that the quality of the architectural design is critical for the successful development of an embedded system. Decisions made in the architecture design phase have a very large impact on the cost and quality of the final system [45]. An additional difficulty is that quality requirements can often conflict with one another and with economic constraints. Two commonly mentioned reasons are: (i) the architecture design sets the foundation for the successful achievement of quality requirements and fulfillment of limited resource budgets, and (ii) the architecture design helps to deal with the ever-increasing complexity of today's embedded systems. However, when a number of architectures can potentially deliver the same desired functionality of a system, designers are faced with a difficult optimization problem. In such cases, the software engineer must find the architecture that entails minimal development and other life-cycle costs. Therefore, the goal of our research is to develop a tool that searches for optimal solutions in potentially large design spaces.

Research Process: We started by looking into the structure of AADL, its hardware and software components. Then we defined the measures between the hosts and components: frequency of interaction, message size, and network reliability, delay, and bandwidth, as well as the calculation of the optimal distribution value. Finally, we implemented the distribution tool [46].

Results: In this research, we present the tool ArcheOpterix that aims to help software architects with the difficult task of selecting an optimal architecture design, with respect to given criteria. ArcheOpterix is an Eclipse plug-in that provides a platform to implement different architecture evaluation and optimization algorithms. To validate the tool a specific multi-objective, multi-constraint component deployment problem has been used. For this problem, similar to the work of Gerard et al. [47], we use two standard quality metrics (data transmission reliability and communication overhead) and three constraints (component location, component collocation, and memory consumption).

Limitations and Future Work: The first experimental results gained from an early implementation of an evolutionary algorithm are encouraging. As a first test case, 4 hosts and 10 components were chosen. The average fitness of the population improves during the iterations and reaches near Pareto-front line. Likewise, we also gain a near optimal trade-off between data transmission reliability and the inverse of communication overhead. However, there is still room for future improvements and there are several interesting research questions to be solved. Since ArcheOpterix should also serve as an experiment platform for optimization algorithms, the tool will be extended with additional optimization heuristics. The performance of these heuristics can then be compared based on their efficiency for benchmark problems. Furthermore, optimization algorithms that can find a good variety of solutions should be implemented.

3.5 Questions Revisited

In papers A to D (described in Section 2) we have addressed the research questions of Section 2.2. In this subsection, we show how the research questions are answered by the corresponding papers.

In paper A, we have addressed research question 1, that is, how to formally describe AADL in an easily implementable way, by defining a denotational semantics for a subset of AADL with Behavior Annex. In paper A, we have also addressed research question 2, that is, how to support the analysis of AADL descriptions in order to guarantee that an embedded system architecture satisfies given requirements. We do this by providing means to perform CTL model checking of AADL models. In paper B, we also addressed research questions 1 and 2 by implementing a tool based on the semantics of paper A. The tool performs model checking on properties defined in CTL, that is, safety and reachability properties.
In paper C, we have addressed research question 3 on how to model timing behavior of components and subsystems in AADL. We have annotated AADL with timing intervals, have written a parser that translates the system into TASM format, where it is possible to perform simulations and to further translate the code into the UPPAAL model checker.

In paper D, we have addressed research question 4: how to deploy a system architecture in a way near optimal. We implemented the tool ArcheOpterix that deploys components onto systems using an evolutionary algorithm. In Table 3.1, one can see the overview of the papers contributions to the research questions.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Research Question</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: How papers A-D contribute to the Research Questions.

All four questions have been at least partially answered, in the sense that we have provided one possible answer to each question.
Chapter 4

Research Method

Different research methods are suitable for different settings, and similarly different validation techniques are suitable for different types of results. The methodology of our research is based on the work of Mary M. Shaw [48] and includes the following research steps:

1. Identification of the research problem from real-world software design and implementation issues. In Chapter 1, we have recognized the need of a formal analysis framework for AADL and its Behavior Annex.

2. Transferring the problem to a research setting, and defining the research questions; in this stage, the research problem is refined and narrowed down. The research problem and research questions are presented in Chapter 2.

3. Answering the research questions and presenting the research results. This stage includes several iteration steps, such as observations, discussions, analysis, and improvement of the research results. The research results are presented in Chapter 3.

4. Checking whether the research results adequately answer the research questions. This can be performed in several different ways, e.g., by implementation of a prototype or by performing case studies. In all four papers presented in Chapter 3, an implementation has been made that has been validated on a case study.

5. Validating the research results in the sense of checking whether they are feasible for the real-world software problem. Since the research done so far does not yet offer a complete solution to the research problem (a thorough validation of the results is missing), the research results have been applied on simple yet relevant case studies presented in the papers of Chapter 3.
Chapter 5

Related Work

This chapter relates the contributions presented in this thesis to relevant research and practice areas, subdivided into three sections. As the papers contain more related work, we here give a short summary.

5.1 Formal Semantics of AADL

The approach we feel is closest to the denotational semantics of paper A is proposed by Ölveczky et al. [49]. The authors have defined a translation semantics from AADL into their object-oriented language Maude, which includes components, port connections, and the Behavior Annex. The AADL components and their subcomponent instances are translated into Maude classes and objects. Maude is capable of simulations and model checking against Linear Temporal Logic (LTL) [50] properties of embedded systems models. In comparison, by employing our AADL semantics, introduced in paper A, one can perform CTL model checking of AADL models of embedded systems.

An approach that is also close to ours is the formal semantics defined by Bozzano et al. [51]. It is centered around the concept of components. For each component, its type, interface, and implementation are given. The component interaction is described by a finite state automaton [36], and the work includes model checking. However, it does not cover elements of the Behavior Annex, as opposed to the semantics of paper A.

5.2 Tools for Model Checking AADL

There are some interesting tools for model checking of AADL and its annexes. BIP (Behavior Interaction Priority) [52] is a language with a formal underlying semantics for description and composition of components, as well as for analysis and code generation of models. Chkouri et al. [53] describe a general methodology for translating AADL and its Behavior Annex specifications into BIP, which allows simulation of systems specified in AADL and application of formal verification techniques developed for BIP. BIP supports model checking of component interaction; however, it does not support model checking of the AADL Behavior Annex.

The Correctness, Modeling, and Performance of Aerospace Systems (COMPASS1) project is an ambitious attempt to fully capture the system under development by applying model-checking techniques. The COMPASS toolset has been developed by Bozzano et al. [54]. It is a tool that models both the nominal and faulty behavior of AADL, and it has been validated in several case studies. The formal foundation of the tool is a semantics by Bozzano et al. [51]. The verification process of the tool is based on NuSMV2, a tool that performs model checking of Finite State Machines [55] and CTL properties. However, in contrast to the tool of paper B, COMPASS focuses on the error behavior of an AADL model. Moreover, the tool

1The COMPASS project is presented at http://compass.informatik.rwth-aachen.de.
2The NuSMV notation is described at http://nusmv.fbk.eu.
is equipped with a powerful and pedagogic graphical interface that, however, is not based on Eclipse and hence is not integrated with Eclipse or OSATE based tools.

5.3 Architecture Deployment

Papadopoulos [56] provides a novel technique and tool support for safety and reliability analysis of automotive software. Their approach consists of system modeling in Matlab Simulink [57, 58], their own tool for fault tree analysis, and genetic algorithms to support the decisions of “whether” and “when” redundancies are needed. Being able to link the system with Matlab Simulink, the authors provide the opportunity for enormous standard analysis capabilities. Our work presented in paper D, gains a similar advantage from the modeling perspective. However, our work is based on AADL and our tool includes the attribute evaluation module.

The DeSi tool, developed by Mikic-Rakic et al. [59], presents a tailored environment for specification, manipulation, visualization and attributes evaluation of deployment architectures. The tool has been developed as a stand-alone Eclipse application and gives an exciting feature of runtime monitoring of live systems by middleware support. The tool is highly useful in the context of deployment architecture optimization but requires that model specifications comply with the format proposed by the authors. In comparison, we provide similar analysis capabilities, by using the standard AADL as the input language to our tool that we have developed as a plug-in to OSATE. By this, we extend its use, in the context of architecture optimization, beyond deployment decision making.
Chapter 6

Conclusions and Future Work

Even though AADL is a powerful architecture description language, it still lacks a formal semantics. We have addressed the limitation by defining a denotational semantics implemented in ML, and encapsulated in an Eclipse plug-in. These achievements make it possible for an engineer to prove features without knowing the details of model checking.

Moreover, AADL does also lack real-time attributes such as time annotations; we have addressed the problem by adding time annotations to AADL and its Behavior Annex, and implemented a tool that translates time-annotated AADL with its Behavior Annex into TASM, which provides the possibility of simulating real-time features. Finally, we have addressed the problem of component distribution by implementing a tool that performs near-optimal distribution.

We have exemplified the applicability of the results through some relevant case studies. However, full validation of the research results in more realistic case studies is subject to future work.

6.1 Contributions

The main contributions of the presented research are summarized as follows:

The Denotational Semantics for AADL and its Behavior Annex. The denotational semantics formally and unambiguously defines a subset of AADL made up of system, system implementation, features, subcomponents, and connections as well as the Behavior Annex. It also includes model checking with properties defined in CTL. Finally, the semantics has been implemented in Standard ML.

The ABV Tool. ABV is an Eclipse plug-in that encapsulates the Standard ML implementation of the denotational semantics. It provides a user-friendly graphical interface, where the user inputs the AADL model and the CTL property.

Time-Annotated Architecture Modeling. We have studied ways to time-annotate a system architecture, and to perform simulations of time-annotated system models. We have chosen to time-annotate the Behavior Annex of AADL. AADLtoTASM is an Eclipse plug-in that translates models defined in time annotated AADL and its Behaviors Annex into TASM, where they can be analyzed with respect to timing, and then further translated into UPPAAL for model checking.

Genetic Optimization Methods for Component Deployment. We have studied ways to reach optimal software deployment on hardware units by using genetic algorithms. ArcheOpterix is an Eclipse plug-in that provides a framework to implement evaluation techniques and optimization heuristics for AADL specifications.
6.2 Future Research Directions

There are many possible future extensions of the work presented in this thesis. The current version of the denotational semantics covers a subset of AADL, which can be extended to cover the whole of AADL. For instance, our subset covers connections between components, but not bus-communication. It would also be valuable to analyze the dataflow of AADL models.

The ABV tool performs model checking by constructing the complete state space, so the evaluation algorithm can be improved. In many cases, it is not necessary to evaluate the whole state space. Consequently, in those cases, it is not necessary to create the complete state space. This would result in both a faster execution and less memory allocation.

It would also be interesting to add time-annotations to the denotational semantics, which would provide us with a new set of problems to work with, for instance the state-explosion problem.

The AADLtoTASM tool generates TASM models. The tool can be further developed to generate UPPAAL models, which use a dense time semantics and are amenable to efficient verification algorithms. The TASM specification [5] includes a translation scheme to UPPAAL. Since UPPAAL is a model checker with timing properties, one could model-check a larger class of timing properties for AADL models.
Bibliography


