Modularity by Design for Safety-Critical Software Systems

An Opinionated Approach using AADL, RTSJ and OSGi

Thomas Drießen

Dissertation
for the degree of
Doctor of Natural Sciences (Dr. rer. nat.)

University of Augsburg
Department of Computer Science
Software Methodologies for Distributed Systems

Januar 2019

Software Methodologies
for Distributed Systems

Modularity by Design for Safety-Critical Software Systems

Supervisor:	**Prof. Dr. Bernhard L. Bauer**, Department of Computer Science, University of Augsburg, Germany
Advisor:	**Prof. Dr. Jörg Hähner**, Department of Computer Science, University of Augsburg, Germany
Defense:	March 21st, 2019

Abstract

"Simplicity is about subtracting the obvious and adding the meaningful."

- John Maeda

A sentence which describes exactly the goal of this work.

But why is simplicity a desirable goal in the first place? To understand this, we investigate how Consumer-Grade Software Systems (CGSS) have evolved over the past decades in terms of complexity. A subsequent comparison with Safety-Critical Software Systems (SCSS) shows that these lag behind the CGSS by about 5-10 years. The complexity of these systems increases to such an extent that they are no longer controllable with the previous methods, languages and tools and therefore new methods, languages and tools were invented to master this complexity again. Hereby, simplicity is the groundbreaking goal and in most cases this simplicity is achieved by abstraction and encapsulation.

Therefore, in this thesis we show how to build on the observed developments in CGSS to bring about improvements for SCSS. We use an oppinionated approach which already specifies the methods, languages and tools to use in order to provide a better overall workflow and developer experience. We show how to start with a semantically well-defined modeling language to generate standard-compliant high-level language code for SCSS. Thereby, reducing the time spent for recurring and error-prone tasks regarding the writing of timing-, structure-, and communication-related code. The only task a programmer then is left with, is the writing of business logic which corresponds exactly to the aforementioned quote about simplicity.

Subsequently, we add modularity by design on top of the aforementioned approach in order to provide better maintainability of the generated systems. This is achieved by reusing an existing modularity framework on top of the chosen high-level language. Thereby, enabling a developer to also use additional benefits of this framework, like predefined, modular services and a well-defined lifecycle that can be reused to achieve even more abstract goals, like mapping modes of the modeling language to runtime reconfigurable modules in code.

Finally, we complement the existing approaches with additional compile- and runtime checks in order to enable a developer of a SCSS to easily show semantic equivalence of exchanged or newly added modules to existing modules in a generated system.

All approaches are evaluated via a qualitative and quantitative evaluation. The qualitative evaluation uses a running example based on a real-world quadrocopter autopilot, whereby the quantitative evaluation makes heavy use of Java Microbenchmark Harness (JMH) benchmarks in order to compare our approaches with a handwritten solution as well as comparing the approaches among themselves.

Acknowledgements

In the time it took to write this thesis, I had the pleasure of meeting a variety of people who all had an influence on the final state of this work. Here is the place where I would like to thank these people.

First of all I would like to thank my supervisor Prof. Dr. Bernhard Bauer for allowing me to write this work in the SMDS lab at all. Thank you, Bernhard, for the many conversations, your advice and motivation. Without them the work as it is now written here would not exist.

Many thanks also to Prof. Dr. Jörg Hähner for agreeing to review this dissertation as a second reviewer. After having corrected several written works myself during my time at the chair, I know what an effort it is. Especially if you haven't seen the work before and it's over 200 pages long. So thanks for that.

In particular I would also like to thank my colleagues at SMDS lab for all the instructive and especially funny years with them. My internships in the years before have taught me that such a wonderful working environment is not a matter of course and therefore I appreciate this time with you all the more.

Last but not least I would like to thank my family and my girlfriend. Mom and Dad: I thank you for everything you have given me. Without your education and the opportunities you offered me, I would not have come as far as I have today. You are the best parents one can wish for. Julia: I would especially like to thank you for having always given me courage in the last phases of my dissertation and for simply being by my side during this time.

Contents

Part I.

INTRODUCTION AND FOUNDATIONS

1 Introduction

1.1. Motivation

Complexity in software is ever-rising - this is an undeniable fact in software engineering. The rise of complexity in software has first been officially recognized in the mid 1960s, when costs for software exceeded the cost for hardware for the first time in history. The term "software crisis" came up at this time and was coined by attendees at the first NATO Software Engineering Conference in 1968. Since then programmers ever have tried to keep in step with the fast-paced evolution of software whose complexity rose exponentially over time. However, how exactly does one measure the complexity of software? In this work we decided to take on a rather simplistic approach and used the Source Lines of Code (SLOC) of a given piece of software under the assumption that every SLOC is fulfilling (partly) a requirement. This assumption might not apply to all software systems, but definitely holds true for Safety-Critical Software Systems (SCSS) where every SLOC usually has to be traceable to an initial requirement. If a SLOC cannot be traced back to a requirement it has to be removed from the code base as it is not fulfilling any purpose.

Therefore, we can prove the rise in complexity by the size of SCSS over time as depicted in Figure 1.1. This graph shows the SLOC in millions for United States Air Force (USAF) aircrafts of the last three decades. The curve implies that SLOC double about every 4 years which implies a quadratic rise in complexity.

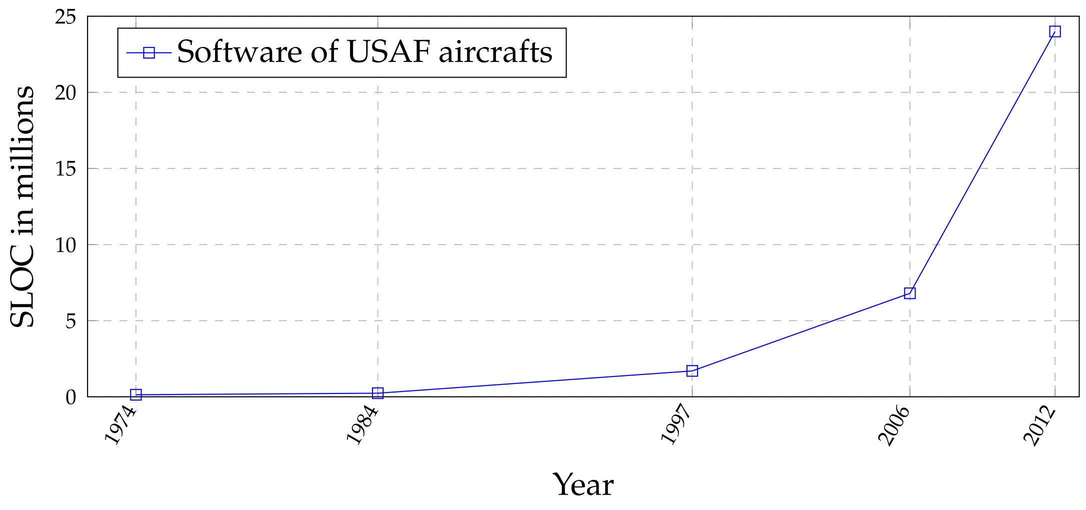

Figure 1.1.: SLOC in millions of USAF aircraft software. [1]

Complexity of software is also always an indicator for costs and time spent on the development of such a system. Given a SLOC in a SCSS then this line was part of a design that has been made and someone had to write and finally test it. The more SLOC a piece of software contains, the more time a programmer spent with writing and testing this code. In addition, one can assume that the more code a project contains, the more design work had to be done beforehand. Additionally, for each SLOC additional documents and reviews have to be created in order to prove compliance with standards and finally being approved by an official authority. Thus, there is a direct connection between number of SLOC, complexity, time and money spent on a given SCSS project. Given this connection between complexity, time and money we have to take measurements to cope with rising complexity in the future in a way that still enables us to develop SCSS.

In this context we took Consumer-Grade Software Systems (CGSS) into account which empirically evolve faster than their safety-critical counterparts, as they are not forced to show compliance with any safety standards. CGSS have undergone a similar process regarding rise in complexity as the one observed in SCSS. In order to illustrate this fact Figure 1.2 exemplarily depicts the evolution of SLOC in millions of the Windows NT operating system over a timespan of 12 years. Starting in 1991 with Windows NT 1.0 (Windows 3.1) and approximately 4-5 million SLOC and ending with Windows NT 5.2 (Windows Server 2003) in 2003 with 50 million SLOC, this figure depicts the same quadratic rise in complexity every 4 years as we already observed in Figure 1.1 for SCSS. Comparing those two charts, we can immediately see that the level of complexity reached by the software of the F-35 in 2012 has already been reached (and even surpassed) by Windows NT 5.0 in 1999. This obvious similarity led us to the assumption that the development of SCSS lags behind about 10 - 15 years compared to CGSS. Therefore, we investigated by what means the developers of these CGSS handled the rise in complexity, costs and time and what tools, languages and processes were used to get a grip on the problem.

A general trend that we could identify is the increasing use of abstraction. In programming languages this trend is reflected in the change of programming languages and paradigms over time. Starting with machine languages like the Z80 being taken over by assembler languages which then again have been overtaken by procedural programming languages like C. That all culminated in the takeover of object-oriented languages in the late 1990s, like C# or Java, which still dominate the current market [4] and thus, are considered current state of the art.

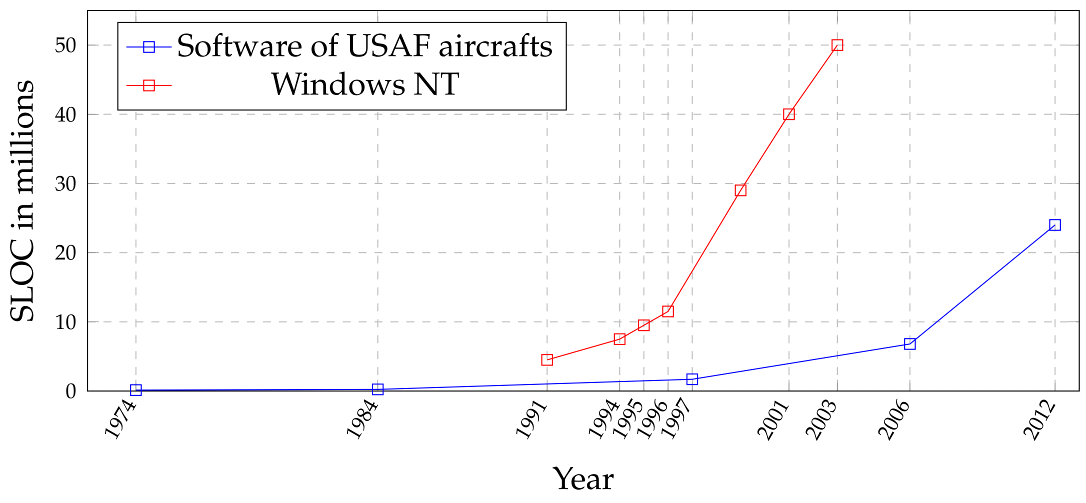

Figure 1.2.: SLOC in millions of USAF aircraft software [1] compared to SLOC of Windows NT. [2, 3]

Another emerging trend was the offspring of model-driven techniques. Starting with simple models of a system to gain an overview, to code generation facilities over to full-fledged low-code platforms like Appian [5] that enable a user to create applications without the need to write a single line of code by himself. Formerly called model-driven development in general, this technique today is sold as so called "low code platforms" [6]. Model-driven software development usually encompasses a modeling language, e.g., the Unified Modeling Language (UML) as source language and a high-level programming language like Java or C# as target language. Thus, by using a model, that in terms of abstraction is even above current high-level programming languages, the task of creating a working system is reduced to drawing a few model elements instead of writing several hundred or even thousand lines of code.

The last trend that could be observed just recently is the shift from monolithic systems towards microservices and a more modular design of applications in general. Each microservice usually defines a Representational State Transfer (REST) Application Programming Interface (API) through which it communicates with other microservices. Higher functionality is provided by combining and orchestrating several of those microservices, e.g., the functionality "open a bank account" is realized by combining microservices for "capture customer data" and "check credit-worthiness". Advantages provided by microservices are - but are not limited to - reusability, e.g., "check credit-worthiness" could be reused in another context, and simplicity because of the reduced size of provided functionality. A drawback is the increased distribution which introduces common problems of highly distributed systems like concurrency or data loss because of network outages.

The remainder of this part is structured as follows: section 1.2 elaborates on the aforementioned major challenges and identifies the accompanying problems of each. In section 1.3 we present the languages and framework we use in this thesis as well as a reasoning on why we chose them. section 1.4 then identifies objectives to deal with the problems and challenges of the design and implementation of SCSS as outlined in the section before. Each objective is paired with the according contribution of this work. section 1.5 enumerates the distinct papers and publications which contributed to this work and have been published beforehand. section 1.6 provides an overview of the entire following work. Finally, section 1.7 explains the typeface conventions we made in order to enable unambiguous descriptions of highly ambiguous sections.

1.2. Problems and Challenges

This section describes specific problems in current performance-critical system development, initially identified by means of literature research and intensive discussions with industrial partners. Based on problems common developers of SCSS face in their day-to-day work when dealing with outdated technology stacks, languages and tooling, we define challenges that will be tackled by this work.

1.2.1. Early Errors and Late Discovery

Although software nowadays permeates our daily lives as never before and more code is written daily, the fact depicted in Figure 1.3 still holds true: All software projects suffer from the early errors and late discovery of those. Usually 70% of errors are introduced during the design phases of a software development process (in this case the V-Model) whereas roughly 80% are not found until the last three phases.The costs of fixing one of those thus, rises from 1 times to 5-80 times the cost if they had been found during design time. These costs even add up when the system under development is a SCSS as usually the work to be done for such a system is many times the amount of a normal, non-critical business software.

Problems The problems resulting from this statistics are diverse:

- Costs of maintenance and operations often exceed costs of development itself
- More time is invested into discovery of errors than writing the software
- Delivery is delayed

Challenge 1 *Error reduction: Provide a methodology for SCSS, which enables developers to reduce common errors and enables them to discover uncommon errors during early stages of the development process.*

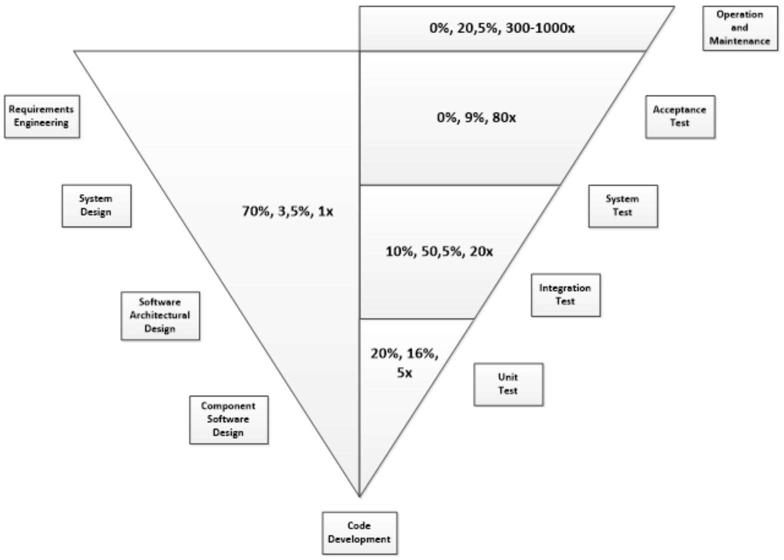

Figure 1.3.: Share of errors introduced versus their share of their discovery across development phases and the relative costs compared to errors discovered during design phase. [7, 8]

1.2.2. Decreasing Maintainability

Figure 1.3 shows that, although not really a phase of development, still a vast amount of 20,5% of introduced errors are found during operation and maintenance which makes them the most costly ones with 300-1000 times as expensive as if been found during design phases. Thus, maintainability is a goal of every SCSS in order to reduce the aforementioned costs as much as possible, especially when the developed system is an embedded one, e.g., cars, planes or spacecrafts, which is usually not as easily reachable as other systems.

Problems Mainly there are three problems when it comes to the maintainability of SCSS:

- The system may not be restarted or otherwise be disrupted by the update process

- Other parts of the system may not be affected by the update and therefore should stay the same as before the update

- Systems shall be easily extendable which assumes the architecture of the system to be extendable by nature

Challenge 2 *Maintainability: Provide a possibility for developers of SCSS, to enhance maintainability of the final system by design and enabling hot updates of running systems while assuring the rest of the system not to be affected.*

1.2.3. Partial Certification

SCSSs are usually subject to certification processes. Certification of new systems usually is a cost- and time-intensive process. When it comes to updating such systems, developers mostly want to certificate the update and not recertificate the whole system which would not be economically feasible. In order to enable such a partial certification one has to prove that the newly added or updated parts are semantically the same as the ones before and also that there are no unwanted impacts on the overall system.

Problems The problem identified by us is:

- The update or new part of a system should be (semi-)automatically provable to be semantically equivalent to the replaced/updated part

Challenge 3 *Partial Certification: Enable developers of SCSS to easily show semantic equivalence of their updates in order to easen the certification process for partial updates of existing systems*

1.3. Technology Decisions

As explained in section 1.1, SCSS and the development of such do lag behind CGSS approximately 10-15 years. Therefore, we assume the aforementioned trends, i.e., increasing use of abstraction, model-driven techniques and microservices, will be of interest for the future development of SCSS. Each trend led to multiple implementations or languages and consequently for each trend we must decide which implementation to choose. In the following we will explain which implementation or language we did choose and why.

The first choice encompasses an appropriate modeling language. Given the fact, that the field of modeling languages is a crowded one, we decided to narrow this field down by defining two premises. First, we are not going to reinvent the wheel, so it has to be an existing modeling language which is in a stable state. Second, as we are settled on the safety-critical domain, the modeling language shall be established in at least one subdomain of safety-critical software development. The first premise excludes all emerging modeling languages and leaves us with the well established ones like UML or Systems Modeling Language (SysML). The second premise cuts down on those even further, effectively leaving us with the choice between Architecture Analysis and Design Language (AADL) and Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) in combination with Autosar. The main difference between those two languages are the well-defined semantics that are declared in the official AADL standard documents. These precisely defined semantics are a distinct advantage of AADL over EAST-ADL regarding code generation. In AADL every modeling element is described exactly in terms of behavior which makes it easy to generate code that behaves accordingly. EAST-ADL in contrast lacks such a precise semantic meaning of its model elements and thus, allows space for interpretation which is neither intended by the authors nor expedient regarding the model-driven approach of this work. Therefore, we decided to go with AADL as the best fitting modeling language for our approach.

The second decision concerns the high-level programming language that shall be used as target language. According to [9] who examines the fitness of programming languages for use in SCSS, there would be only one choice, i.e., ADA with a fitness of 93%, directly followed by Java, although with a difference of 21%. However, there are other criteria that can influence the choice of language in large projects like commonness or popularity. According to [10] ADA is on place 24 of the most popular programming language at the time of writing with an estimated market share of 0,87%, whereas Java constantly resides among the top 3 programming languages since 15 years with an estimated market share of 16,38%. Additionally, the original classification of [9] only examined a pre-1.4 version Java and also not the Real-Time Specification for Java (RTSJ) version of it. If we take into account the changes made to Java since then, we can lift the fitness of Java for SCSS up to 80% instead of 72% according to the criterias defined in [9]. This fitness combined with its popularity let us choose the RTSJ version of Java as our high-level programming language, i.e. target language.

Lastly, we needed means to modularize the code that is produced by our transformation from modeling language to high-level programming language. As we decided to go for Java as target language, we only had the choice between two module systems. The first one is project Jigsaw which was introduced in Java 9 and, at the time of writing, still struggles to claim its place in the enterprise world. The second one is OSGi which looks back on an over 15 year long history and is a well-established framework for modularizing Java code. Beyond the modularization of code it provides a standard specification which specifies several services that might be of use in a future system. Eventually, the choice was made in favor of OSGi as it simply poses the more mature framework although project Jigsaw is the new default in Java 9.

All of the above mentioned technologies are meaningless if not utilized in an expedient way. Thus, we will define a methodology which combines the best of those technologies in order to partially tackle the 3 major identified challenges of SCSS development, i.e., early errors and late discovery, maintainability and partial certification.

Besides these major problems, we also expect to reduce more common low-level errors like buffer overflows or reduce the time spent with recurring tasks like garbage collection by using the aforementioned languages and framework.

1.4. Objectives and Contributions

This section identifies objectives to deal with the problems and challenges of the design and implementation of SCSS as outlined in section 1.2. Therefore, we provide an overview of identified objectives as illustrated in Figure 1.4 and list the main contributions of this thesis. To achieve these contributions our approaches utilize common techniques known from Model-Driven Development (MDD) and high-level languages as well as software stacks built upon and around them in order to extend the boundaries of current best practices.

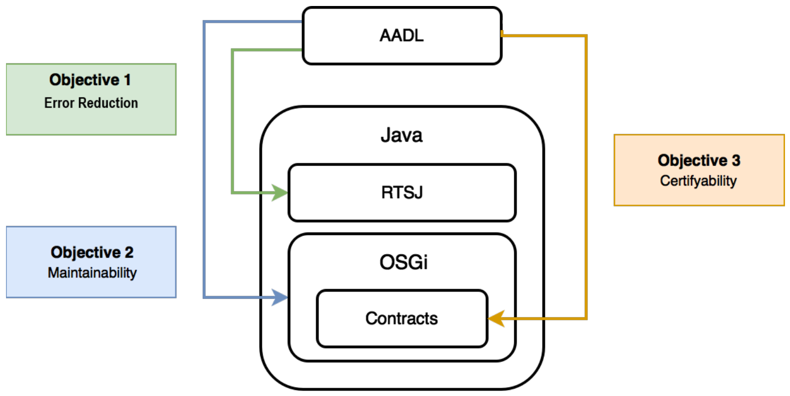

Figure 1.4.: Objectives overview

1.4.1. Error Reduction

Software in SCSS nowadays usually is written by hand using rather mature languages like Assembler or C. This results from the need to verify and validate the written code during unit or integration testing as well as during reviews done by authorities which are necessary for the often aspired compliance to certain standards, e.g., DO-178B/C in aviation or ISO-26262 in automotive industry. During such reviews handwritten code usually is better validable than generated instruction that stem from higher level languages and their compilers. Thus, modern techniques like MDD or high-level programming languages like Java are not widely used in the domain of SCSS although providing added value in terms of error reduction and frontloading of technical debts. Therefore, in our first objective we aim to shift stakes in favor of the aforementioned techniques in order to give an example of their value in the domain of SCSS. Figure 1.4 depicts several core technologies around which our overall approach is centered. At the core there is AADL which poses the starting point for any SCSS design. Its centralized architecture model enables us to capture all relevant information needed for the automated transformation into several aspects of the implementation which is based on Java, more precisely RTSJ. Those depicted parts are used to overcome Challenge 1.

Objective 1 *Use MDD techniques in combination with a semantically well-defined modeling language and a capable, modern high-level target programming language in order to lower coding errors via code generation and usage of a managed, high-level programming language as well as enable early detection of errors during design phase via model analysis.*

Contributions In this work we reuse AADL with its well-defined semantics as well as RTSJ with its industry-proven capabilities. In order to diminish the number of errors introduced in the design phase as shown in Figure 1.3.

- We contribute a mapping from a subset of AADL to RTSJ that preserves the semantics of all utilized AADL model elements while leveraging the facilities of RTSJ as an high-level programming language, e.g., automatic memory management, garbage collection, no pointer arithmetics and runtime checks.

- We created a generation facility that is able to take any sufficiently detailed AADL models and turns the structure as well as communication patterns and timing constraints into RTSJ code.

AADL and its ecosystem of tools already offer means by which a developer can verify the system from different viewpoints, thus reducing the number of errors left in the system model significantly. Our contribution enables the developer to generate a skeleton application which adheres to the system design choices made and its semantics, proven correct by the already available tooling.

1.4.2. Maintainability

Although, Objective 1 and our contributions in that regard already can offer a benefit in terms of error reduction over traditional development techniques and languages, Challenge 2 remains largely untouched. Each change that is introduced to a system generated with our approach still leads to the whole system being regenerated and by this rendering existing components of the old system non-reusable. In terms of maintainability this poses an untenable drawback that would render the whole approach obsolete. Therefore, we tackled Challenge 2 by altering the former approach to also take maintainability of a generated system into consideration. As we decided to use Java and RTSJ as target language, the possible technical candidates to tackle Challenge 2 have been narrowed down to only a few. These lasting ones need to be able to replace code in running systems as well as providing mechanisms that resolve dependencies between different parts of the software. We decided to facilitate OSGi, a component framework built on top of Java, as a target framework for the generation facility of subsection 1.4.1, to incorporate the second challenge into our existing approach.

Objective 2 *Enhance the mapping of subsection 1.4.1 to target a software stack that enables developers to implicitly design and generate runtime reconfigurable and interchangeable software components, thus enhancing overall maintainability*

Contributions In this work we utilized the capabilities of AADL and OSGi to provide the following contributions:

- We introduce a mapping from AADL to RTSJ that simultaneously delivers reconfigurable and interchangeable software components by leveraging AADL's inherent component-oriented design and transferring it onto semantically equivalent concepts in the target framework, i.e., OSGi.
- We combine several of OSGi's capabilities in order to enable a hot deployment of updates to a running SCSS and also to easen the extension of existing SCSS without compromising the previous existing compliance of such a system.

By reusing OSGi we also harness its vast tooling ecosystem that leads to increased developer productivity and also to reduced error introduction, thus amplifying the effect achieved through the contributions of Objective 1.

1.4.3. Certifiability

Objective 1 and 2 as well as our contributions in that regard, provide a framework that is capable of generating a working and maintainable skeleton system from any sufficiently detailed AADL models. Nevertheless, the generated, self-contained components still are missing additional information to make them (semi-)automatically provable in regard to semantic equivalence and therefore standard compliant interchangeability which is largely the demand of Challenge 3. Therefore, means are provided to easen the prove of compliance of newly created software components or updates of existing ones. This is achieved by enhancing the existing generated code base with additional metadata in form of requirements and capabilities that enable a developer to automatically show equivalence of components to a certain extend. Also a mechanism is provided to enforce runtime contract enforcement.

Objective 3 *Easen the creation of reusable and interchangeable software components whose semantic equality can be (semi-)automatically shown and enforce their stated contract at runtime*

Contributions To enable the generation of reusable and interchangeable software components whose semantic equality can be (semi-)automatically shown we developed the following artifacts:

- We created a range of capabilities and corresponding component property types to capture contract specific information.
- We reused mechanisms provided by OSGi to automatically show the semantically equivalence of software components through the process of resolving.
- We leveraged existing mechanisms in OSGi to enforce arbitrary contracts between components at runtime.

1.5. Publications

In this section we present published work concerning this thesis, thereby clarifying our own contribution within it and its relation to the parts of this work.

- Driessen, T., & Bauer, B. (2016, September). Shifting temporal and communicational aspects into design phase via AADL and RTSJ. In Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th (pp. 1-10). IEEE. [11]

This paper primarily describes parts of the approach presented in chapter 3, i.e., the mapping approach from AADL to RTSJ. This is solely our own work.

- Driessen, T., Bauer, B., Honke, B., & Kuhnmünch, M. (2015, September). Layered-V. In Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th (pp. 10A2-1). IEEE. [12]

This paper was in parts reused for the basic section presented in chapter 2 and also is solely our own work.

1.6. Outline

This section describes the outline of this thesis. Depending on the reader's technical expertise chapter 2 can be skipped entirely or at least partially. However, it is strongly recommended to read chapter 3, chapter 4 and chapter 5 in sequence as the results and some other parts of the respective chapter are assumed to be known in subsequent chapters. Each chapter starts with a motivation for its respective focus of attention, explains additional basic knowledge if necessary, then elucidates the respective contributions, compares those with related work and finally evaluates them directly within each chapter which is why we refrained from adding a separate evaluation chapter at the end of this thesis. Finally, chapter 6 and chapter 7 provide an overview of possible further work that could be done to enhance our approach as well as an outlook to what extend this approach could be used for designing and implementing truly modular SCSS in the future.

Chapter 1 provides an introduction and motivates our thesis. It details our aimed application environment, for which we identified problems and challenges. Based on that, we derive objectives to be faced in this thesis and list our contributions.

Chapter 2 provides necessary background information about technologies and techniques that are used throughout this thesis. As the main contributions can be allocated to the research area of model-driven development of safety-critical, service-oriented software system, the used source language AADL and the used target language RTSJ are explained in detail as well as the used framework for service-orientation and modularization on top of RTSJ, namely OSGi.

Chapter 3 details our mapping approach from a sufficiently detailed AADL model to RTSJ which maintains the semantics given by the AADL standard in order to enable developers of SCSS to shift structure, timing and communication-related concerns into design phase. Hence, enabling them to perform analyses regarding communication and timing during design phase while resting assured that the implementation will reflect their design choices. The application of this approach is shown via the implementation of an autopilot for quadrocopters, once handwritten and once generated by the implementation of the presented approach.

Chapter 4 details the adoption of chapter 3's approach, targeting not only a high-level language as target language for code generation, but also a modular code base by design. Thus, enabling developers of a SCSS to not only shift structure, timing and communication-related concerns into design phase, but also to create a highly modular, runtime reconfigurable and, most important, a more easy to maintain systems by design. Advantages explained in the former approach still hold true, i.e., being able to perform different analyses and resting assured that the implementation will reflect AADL semantics while new advantages, i.e., enhanced maintainability and better configurability are added on top. The evaluation of this approach shows advantages as well as drawbacks of this approach, e.g., better performance compared to chapter 3's approach, but at the same time additional computing cost as well as an increased memory consumption during reconfiguration of a system.

Chapter 5 presents an extension to the two aforementioned approaches in terms of semantic restriction of components. By building on top of OSGi's requirement/capability model four different capabilities are provided that enable developers of SCSS to express additional functional and non-functional properties of a service implementation. First, two capabilities for pre- and post-conditions as well as type ranges, i.e., allowed values for the types defined in service `interfaces`, is provided. Second, a capability for hardware dependencies is presented which allows developers to express mandatory dependencies of their software components to specific hardware requirements, e.g., a specific amount of ram or a specific cpu architecture. Finally, a capability is shown that enables developers to define a Worst-Case Execution Time (WCET) for their service implementation. The application of these capabilities is shown by applying them to several small examples taken from the use-case presented in section 2.1. Subsequently, we present an approach how the contracts between components, defined by those capabilities, can be enforced at runtime by leveraging existing concepts of OSGi.

Chapter 6 details three possible extensions that would greatly enhance the usability of the aforementioned approaches in a real-world scenario. First, several additional AADL components, **`features`** and properties are named that would be necessary to broaden the applicability of our approaches enough to be usable outside of a pure research context. For each of them we propose a possible transformation approach and highlight possible future challenges of their implementation. Second, we show how inter-process communication might be incorporated into our existing approach, which currently only supports single processes. Therefore, we provide a glimpse on OSGi's remote service mechanisms which would be a perfect match for an inter-process scenario. Finally, we show which additional capability contracts might be sensible and what has to be done to incorporate our contract approach into the existing generation facility.

Chapter 7 provides an extensive conclusion of our three approaches and provides some thoughts on how the presented work might be used in the future to create a standardized component-oriented development methodology for SCSS.

1.7. Typeface Conventions

In this section we will provide the typeface decisions we made in order to provide a homogeneous and comprehensible look and feel for this work as well as an unambiguous way of describing ambiguous sections within this thesis.

In this work we will make heavy use of code listings, be they for source code or model code. Often those code listings are used to show how to map a concept from a source language into a target language. In those cases it might be confusing when the references from the text to the respective listings contain the same names. Therefore, we decided to type AADL code/concepts/types in a bold, verbatim style, i.e. **`like this`**, whereas Java code/concepts/types and also every other code language is typed in normal verbatim style, i.e., `like this`. In some places we map general concepts from AADL to Java and therefore need a third type of accentuation for those concepts that are neither AADL nor Java specific. In those cases we type general concepts in bold, i.e., **like this**. A bold typeface is also used for important concepts when they are introduced for the first time. At some places we also make use of quotation marks to emphasis certain words.

Whenever one of the accentuated words is used in plural or with an apostrophe, then those are typed in the same typeface as the word itself in order to keep the appearance consistent.

2 Foundations

2.1. Running Example and Evaluation Baseline

In this chapter we introduce a running example that we will reuse (partially) in the remaining chapters for source code and model code listings. This running example is used to offer a practical real-world example in order to be able to better illustrate the matching of abstract concepts of our work on a real-world SCSS.

The running example under consideration is based on an autopilot, developed by students of the practical course "Avionik Praktikum" at the University of Augsburg. The goal of this course was to write a working autopilot for a quadrocopter that is simulated in X-Plane [13]. The autopilot software is running on a separate device – a Raspberry Pi 2 with special autopilot hardware from Erle Robotics S.L. [14] – and communicates with the simulation via UDP messages. The software is written completely in Java, respectively RTSJ, and is running within the JamaicaVM [15] from aicas GmbH on a real-time Linux. The current state of the software used for this use-case encompasses seven different components which are depicted in Figure 2.1.

The first component `simulation` provides means to connect the autopilot code to the mentioned simulation. This component sends `CommandPosition` commands and `InformationPosition` informations to the `positionControl`. `CommandPosition` commands are usually waypoints the quadcopter shall pass and `InformationPosition` informations are the current values for pitch, roll, yaw and altitude, as well as longitude and latitude, that are provided by the simulation. The `positionControl` takes care of converting `CommandPosition` commands and `InformationPosition` information into high-level commands for each of subsequent controls, i.e., `pitchControl`, `rollControl`, `yawControl` and `altitudeControl`. It also forwards the current `InformationPosition` information to those controls.

The controls for yaw, pitch, roll and altitude take the high-level commands provided by `positionControl`, as well as the position information, to convert each of the high-level commands into throttle values for each of the quadcopter's engines. Those are forwarded as `RequestThrottle` requests to the `mixThrottlesControl` which waits for each of the four controls to send such a request before it accounts them with each other to obtain a single `CommandThrottle` command that is sent back to the `simulation`. The `simulation` then internally sends those commands to XPlane and receives new `InformationPosition` information, that are fed back to the rest of the autopilot system. In the handwritten solution each component is currently implemented as an `AsyncEvent-Handler`. All handlers have a **`period`** of 20 Hz or 50 ms, and are hence **`periodic`**. Thus, the messages are **`sampled`**, no **`immediate`** or **`delayed connections`** exist between the components.

Given this starting point, we modeled the same autopilot only using the defined subset of AADL elements defined later in section 2.2. We decided to represent the messages that are exchanged between the different controls via three common super **`data type declarations`**, i.e., **`Information`**, **`Request`**, **`Command`**. Those **`data type declarations`** are then realized by several **`data implementation declarations`** which represent the more specific values of these messages as **`data subcomponents`**, e.g., the message **`CommandPosition`** is mapped onto a **`data implementation declaration`** that contains three **`data subcomponents`**: **`altitude`**, **`latitude`** and **`longitude`**. The three **`subcomponents`** use **`Basic-_Type::Float`** as classifier.

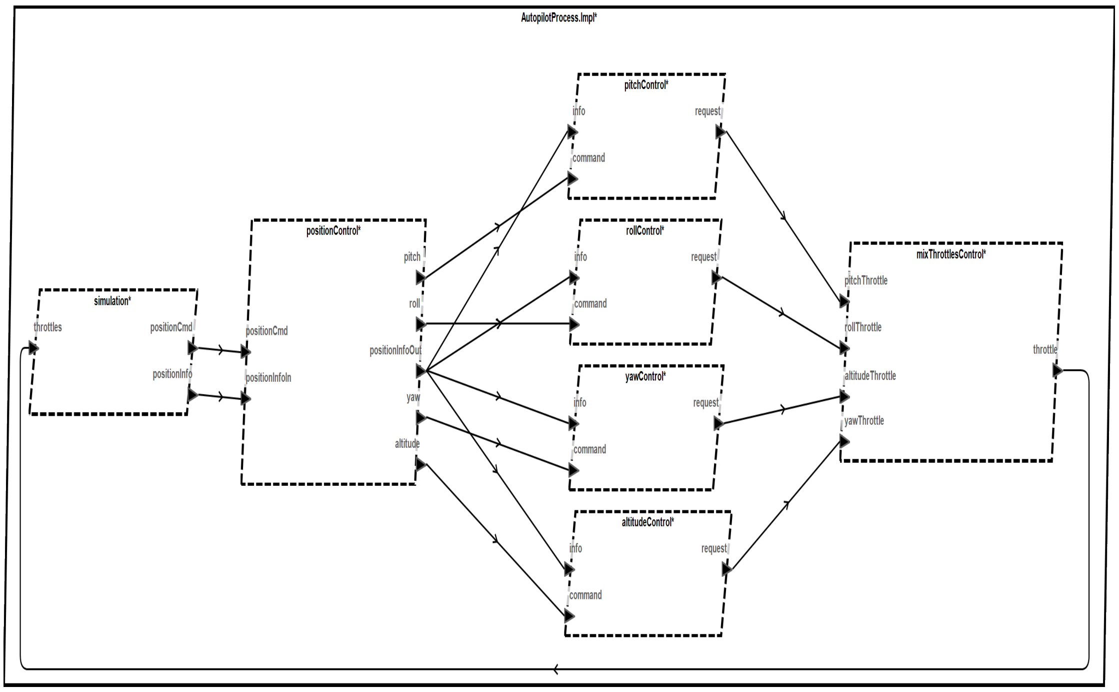

Figure 2.1.: Architecture of the Autopilot in AADL

Subsequently, each component is mapped onto a **`thread`** in AADL as can be seen in Figure 2.2, whereby the four basic controllers – **`Pitch-`**, **`Roll-`**, **`Yaw-`** and **`AltitudeControl`** – are modeled via AADL's **`extension`** and refinement mechanisms. The common parent **`thread type declaration`** is **`PIDController`** that defines the **`features info`** and **`command`** as **`in data ports`** and **`request`** as an `out data port`. Afterwards, we created four type declarations for pitch, roll, yaw and altitude, extending `PIDController` and refining the `command in data port` to its corresponding message type, e.g., `CommandPitch`, `CommandRoll`, etc.. The remaining controllers are modeled as separate `thread type declarations` as they do not share common `features`. Then, we connected each sending component with its receiving counterparts, resulting in the explicit communication architecture depicted in Figure 2.1.

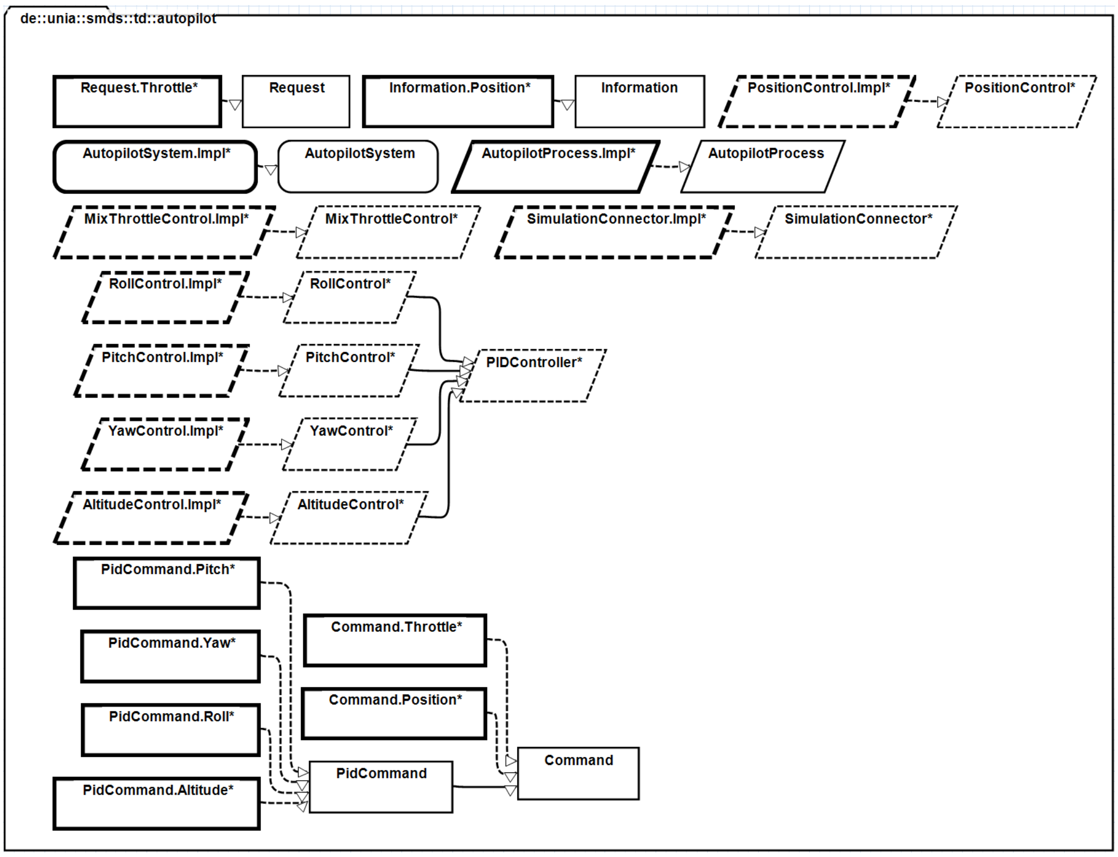

Figure 2.2.: Package of the Autopilot in AADL displaying all components and their hierarchies

The presented running example will not only be used to illustrate basic concepts in the basic section, but also for the evaluation of out mapping approaches. Whenever the running example is not sufficient to explain a certain concept we will fall back to a more general example. Evaluations in this work all are divided into a quantitative and a qualitative part. Within the qualitative part the running example will always be used to show the impact of the respective approach on actions like addition, removal or modification of a component, one of its `features` or the `connections` in between.

The quantitative parts of the evaluations will be done by creating Java Microbenchmark Harness (JMH) [16] benchmarks.

> *"JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targetting the JVM."[16]*

The creation of a JMH benchmark is fairly simple as depicted in Listing 2.1. The benchmark itself is just a method annotated with `@Benchmark`. JMH generates the benchmarks based on the annotations it finds in given `class` and then creates an executable Java Archive (.jar) file. This .jar file then can be executed to perform the benchmark of code given within the annotated method. In Listing 2.1 the method was intentionally left blank, as it is used to create a baseline for all other benchmarks of our work.

```
public class BaselineBenchmark {

    @Benchmark
    public void baseline() {
        // this method was intentionally left blank.
    }
}
```

Listing 2.1: Simple JMH baseline benchmark.

The benchmarks in this work usually produce results in terms of operations per second, which illustrate how fast (or slow) the respective code executes. However, without a proper baseline with which the results can be compared those operations per second are rather meaningless. Additionally, benchmarks tend to produce different results on different hardware. Therefore, we created an empty benchmark that gives us a baseline to compare all other benchmarks against as this benchmarks poses the maximum amount of operations per second that are possible on the given hardware. A baseline produced on different hardware thus, would provides means to compare it with the baseline produced by our hardware. Therefore, a baseline then also can be used to compare the same benchmarks executed on a different machine. This baseline is depicted in Figure 2.3 as a black bar with 2656466293.421 operations per second which is a mean value. The red bars show the error rate or distribution for this benchmark, i.e., how much the results varied during benchmark execution. The variation for our baseline benchmark is +- 28089138.228 operations per second.

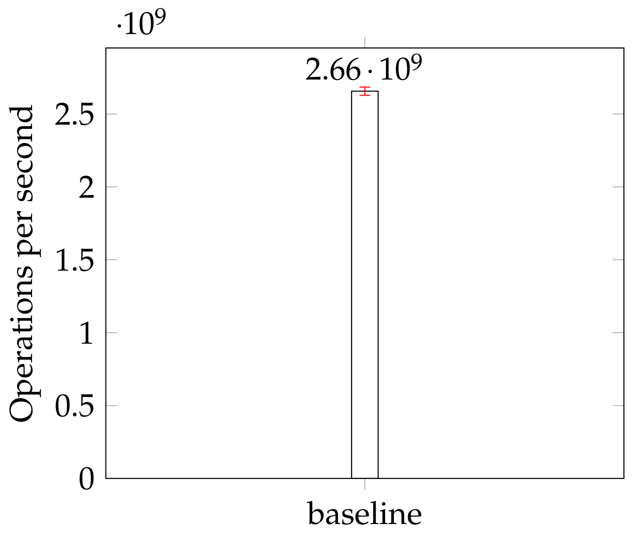

Figure 2.3.: Baseline for all benchmarks

The created JMH benchmarks in this work will be equivalent to the code that would be actually generated by the respective approach, e.g., if the approach would generate a large switch-case statement in order to pass messages from a sender to the right receiver then a corresponding equivalent JMH benchmark would only contain a realistic switch-case statement in terms of size and complexity of code within each case statement. The benchmark would not contain complex sender and receiver `classes` or a surrounding framework, as those might distort the benchmark results. The use of JMH is necessary in order to achieve reliable results, as benchmarking on the Java Virtual Machine (JVM) is a highly complex topic, see [17].

2.2. Architecture Analysis and Design Language

2.2.1. Introduction to AADL

> *"Developed by a SAE International sponsored committee of experts, the Architecture Analysis and Design Language (AADL) was approved and published as SAE Standard AS-5506 in November 2004. Version 2.1 of the standard was published in Sept 2012. The AADL is designed for the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems. It provides a new vehicle to allow analysis of system designs (and system of systems) prior to development and supports a model-based, model-driven development approach throughout the system life cycle."* [18]

AADL promotes the approach of one centralized model and stakeholder-specific views. Thus, all data is centralized and changes in one view affect other views. This way the chance of introducing errors by changing one view and thereby breaking another is minimized.

Although the field of modeling languages is very crowded, AADL excels by defining an official, standardized semantic for all its model elements and properties. This makes it an ideal candidate for model-to-model or model-to-text transformations as the semantics to be fulfilled by the target language are already predefined and standardized by the source language.

AADL provides three main categories of modeling elements: components, **`features/connections`** and properties. Components pose the main building blocks of a system and can be nested within each other with some semantic restrictions. **`Features`** can be added to components and thus define their interface through which they can interact with other components that are connected to them via **`connections`**. Finally, properties can be used to alter the semantics of modeling elements. As AADL is a very complex and powerful language, we decided to restrict our mapping approach to a specific subset. In the following sections we will provide a detailed description of modeling elements that are part of this subset.

2.2.2. Components

In AADL each component is divided into a type and an implementation. A type describes the component in term of **features** that can be seen from the outside and thus providing an interface-like view on the given component. An implementation in turn describes a component regarding its inner composition, i.e., its **subcomponents**, **connections** between those, etc.. An implementation must implement a type, whereas a type must not have an implementation. Both, type and implementation, can extend other types and implementations and specialize them by doing so. This enables users of AADL to create inheritance hierarchies and thus simplify the development of components through reuse.

All components in AADL can be represented in two ways (except packages): A graphical and a textual notation. Those notations will be shown for each element as in parts of this work we use the textual notation (e.g., for comparing model code with generated code) and in other parts we use the graphical notation (e.g., for system overviews).

2.2.2.1. Package

Packages are used to organize sets of components and therefore resemble packages known from different programming languages. As in Java, **packages** in AADL can be used to create hierarchies through naming schemes. **de::unia** would thus be a parent **package** for **de::unia::smds**. In contrast to Java, where packages are usually folders that can contain other folders and hereby creating a hierarchy, AADL **packages** cannot physically contain other **packages** (e.g., defined in code). **Packages** can contain a **private** and a **public** section. **Components** defined in the **private** section are hidden from components in other **packages** and thus cannot be further extended or refined by these. **Components** defined in **public** sections are exposed to other packages which can declare dependencies to them via a **with** clause at the beginning of their **package** declaration as can be seen in Listing 2.2. This mechanism resembles for example Java's import statement.

```
package autopilot
  public
  with Base_Types;
    ...
  private
    ...
```

```
end autopilot;
```

Listing 2.2: AADL Package Declaration

2.2.2.2. Process

"An AADL process represents a protected address space that prevents other components from accessing anything inside the process and prevents threads inside a process from causing damage to other processes. [...] A process specification does not include an implicit thread; therefore, a complete process specification should contain at least one explicitly declared thread or thread group. This process may have source code associated that represents the executable code and data." [19]

In our approach we use **`processes`** as uppermost composite component aggregating all **`data`**, **`subprograms`** and **`threads`** we need for the execution of a given set of tasks. In Figure 2.4 the general graphical and textual notation of a **`process`** component can be seen. As all components have to be declared within a **`package`**, our **`process`** type **`AutopilotProcess`** is declared within **`autopilot`**. In the textual representation an implementation of a type must declare the type, i.e., **`process`**, the keyword implementation and the type to extend via its name, i.e., `AutopilotProcess`. The name of the implementation itself is given after a separating dot, i.e., **`impl`**. All components have to be closed by a final **`end`** keyword and the name of the component. In the graphical representation the implementation is shown via a dotted arrow that points to the type being implemented.

```
package autopilot
    public
        process AutopilotProcess
        end AutopilotProcess;

        process implementation AutopilotProcess.impl
        end AutopilotProcess.impl;
end autopilot;
```

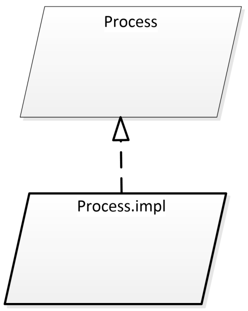

Figure 2.4.: AADL process type and implementation

2.2.2.3. Thread

"A thread represents an execution path through code that can execute concurrently with other threads. [...] The code of a thread executes within the address space defined by the process [...] that contains the thread." [19]

In our approach, and also in AADL, `threads` are the core concept for describing running software. Every piece of software being executed is executed by a `thread`. Therefore, AADL describes the semantic of the lifecycle of `threads` in great detail in terms of timed automaton. It defines different lifecycle states which are reached through lifecycle events. Lifecycle events are for example **dispatch**, **start**, **completion** or **deadline**. Those four lifecycle events are depicted in Figure 2.5.

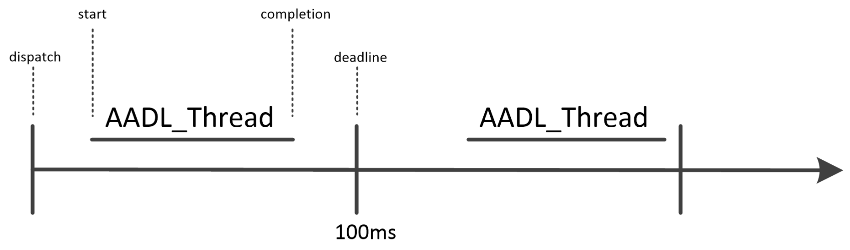

Figure 2.5.: AADL `thread` lifecycle events.

Dispatch marks the beginning of a new dispatch cycle whereas **start** marks the actual start of the computation of a `thread`. The lifecycle event for the end of the `thread's` computation is called **completion** and the end of the dispatch cycle is called **deadline**. In case of periodic `threads` the end of a **dispatch** cycle is marked via the given **period** (which is explained in detail in subsubsection 2.2.4.1) and not via its **deadline**. Also for periodic `threads` the lifecycle event marked as **deadline** in Figure 2.5 is also a new **dispatch**, as periodic `threads` are dispatched recurringly. Figure 2.6 depicts the textual and graphical definition of an AADL `thread`.

The control over dispatching, parallelism of threads or in general scheduling is the task of a scheduler which we consider out of scope for this work as it is also not directly covered by AADL.

The semantics of `threads` can be further altered through properties which will be explained in detail in subsection 2.2.4.

```
package autopilot
    public
        thread PIDController
        end PIDController;

        thread implementation PIDController.Altitude
        end PIDController.Altitude;
end autopilot;
```

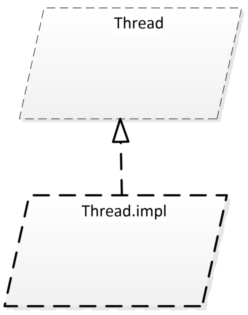

Figure 2.6.: AADL thread type and implementation

2.2.2.4. Subprogram

> *"A subprogram represents a callable unit of sequentially executable code. [...] Subprograms can have parameters through which data values are passed with and returned from a subprogram call." [19]*

In our approach **`subprograms`** are used to model methods that are called by user code, e.g., methods that are executed when a **`thread`** reaches one of its lifecycle states. Therefore we only use the possibility to model **`parameters`** and not the possibility to require **`data access`** which would grant the **`subprogram`** access to global state or shared data. **`Subprograms`** are the only model elements containing user specified code. One can model **`subprograms`** as direct **`subcomponents`** of a **`thread`** component. Another possibility is modeling **`subprograms`** as **`subcomponents`** of **`processes`** and granting a **`thread`** access to this **`subprogram`** via **`requires subprogram access connections`**, which are however not part of this work.

Analogous to **`threads`** and **`processes`**, **`subprograms`** can be defined in textual notation via the keyword **`subprogram`** and a name as shown in Figure 2.7. An implementation is also declared the same way it is for **`threads`** and **`processes`**.

2.2.2.5. Data

> *"Data component types represent application data types. Data component implementations [...] allow you to specify the internal structure of a data type." [19]*

```
package autopilot
    public
        subprogram Compute
        end Compute;

        subprogram implementation Compute.
            AltitudeController
        end Compute.AltitudeController;
end autopilot;
```

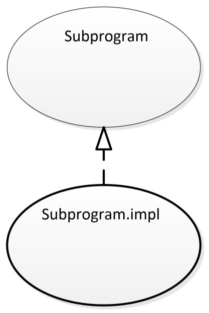

Figure 2.7.: AADL subprogram

`Data` components can be used in different places within a model. First, one can use a formerly declared `data` type in order to describe the data transmitted over a `connection`, `port` or `bus`, whereby only the former two matter for our approach. Second, one can declare `data` types as `subcomponents` within another component. This way we usually describe local or shared data within a `thread` or in a `process`. Third, `data` can be used to describe the parameters for a `subprogram` and thus serve as a way to detail the interface of a `subprogram` component. The semantic of `data` components can be altered by properties which is described in detail in subsection 2.2.4.

Figure 2.8 shows the usage of a user defined `data` type as classifier for the `port` of a `thread`. The `port` is defined within the `features` section of `PIDController`. The name, i.e., `cmd`, is followed by the direction and type of this `port`. Finally, the classifier `Command.Altitude` is given to describe the data being transmitted over this `port`. Within the `subcomponents` section of `PIDController.Altitude` `Command.Altitude` is used as a classifier for the `data subcomponent localData`.

2.2.3. Features and Connections

After describing the basic components we now will have a closer look on how those components can be connected and how the communication between them is defined in AADL.

As previously seen in Figure 2.7 and Figure 2.8 and also already described in subsection 2.2.2 each component type can declare `features` which are considered as an interface of this component. AADL defines different types of `features`, like `event data port`, `event port`, `requires/provides data/subprogram access` of which we only will use `data ports`.

```
package autopilot
    public
        with Base_Types;
        thread PIDController
            features
                cmd : in data port Command;
        end PIDController;

        thread implementation PIDController.Altitude
            subcomponents
                localData : data Command;
        end AADL_Thread.impl;

        data Command
        end Command;
end OurPackage;
```

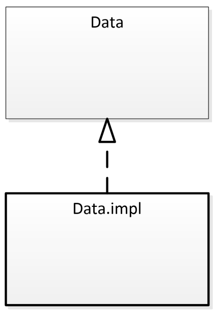

Figure 2.8.: Data as port classifier and local data

2.2.3.1. Data Port

`Data ports` are specialized **`features`** that allow components to exchange data. They can be directional (in/out) or bidirectional (in out). We only consider directional ones as each bidirectional one can be replaced by two directional ones. Usually, the type of data that can be received/sent is declared by a classifier, which is either a user defined **`data`** type as in Figure 2.8 or one of the predeclared **`data`** types provided by AADL, e.g., **`Basic_Types`**. In contrast to **`event data ports`** which can buffer a defined amount of data, **`data ports`** only can hold one data at a given point in time. Incoming data always replaces the current data. In Listing 2.3 the **`thread`** type **`PIDController`** declares two **`data ports`**, one incoming, the other one outgoing, both with an user defined **`data`** type.

```
package autopilot
    public
    thread PIDController
        features
            cmd: in data port Command;
            request: out data port Request;
    end PIDController;
end autopilot;
```

Listing 2.3: In- and outgoing data ports with defined classifiers.

2.2.3.2. Port Connections

`Port connections` are a specific subtype of **`connections`** and can only be declared between two **`ports`**. **`Port connections`** can be directed or bidirectional and thereby indicating the direction of the data flow. Again, we decided to only consider directed **`connections`** as each bidirectional **`connection`** can also be expressed by two directed **`connections`**. Given a **`connection`** is directed, i.e., if a **`connection`** is starting at **`port`** A and ending at **`port`** B and both **`ports`** are declared by sibling components, then A must be an **`out port`** and B an **`in port`**. There are other cases, e.g., **`port`** A being declared by a parent component and **`port`** B being declared by a child component, where A and B both must be an **`in port`**, but the port direction may never be contradicting the data flow direction and vice versa. For two **`ports`** A and B connected via a **`port connection`** AB one can check if the data produced by **`port`** A, transmitted over **`connection`** AB and received by **`port`** B is consistent, e.g., if it is the same **`data`** type. The rules for consistency, as other semantics, can be altered by properties which are explained in detail in subsection 2.2.4.

Listing 2.4 shows an exemplary **`connection`** between two **`threads`**, **`Simulation`** and **`PositionControl`**. Both **`threads`** are contained within a parent **`process`** component, i.e., `AutopilotProcess.impl`, which declares two **`thread subcomponents`** whose classifier reference **`Simulation`** and **`PositionControl`**. **`Simulation`** declares an **`out data port positionCmd`** and **`PositionControl`** an **`in data port`** also named `positionCmd`, both with the classifier **`Command`**. In the parent component **`AutopilotProcess`** those two ports are connected via the **`port connection poscmd`**, declared within its **`connections`** section.

```
package autopilot
    public
      thread Simulation
          features
              positionCmd: out data port Command;
      end Simulation;

      thread PositionControl
          features
              positionCmd: in data port Command;
      end PositionControl;

      process AutopilotProcess
      end AutopilotProcess;

      process implementation AutopilotProcess.impl
           subcomponents
              simulation: thread Simulation;
              positionControl: thread PositionControl;
```

```
            connections
                poscmd: port simulation.positionCmd -> positionControl.
                    positionCmd;
        end AutopilotProcess.impl;
end autopilot;
```

Listing 2.4: Port connection between two threads within a common process.

2.2.3.3. Semantic Connections

AADL differs between its component model and an instance model. The component model is considered to be a collection of reusable building blocks whereby the instance model is one specific manifestation of a combination of such components. This has consequences for the `connections` between different building blocks. Figure 2.9 shows the component model of a `system` implementation which declares several different `subcomponents`. As section 2.1 does not define a system with two levels of hierarchy which we would need to explain a semantic connection, we define an extra example as follows:

A `system` type with corresponding implementation called `ComplexSystem` and `ComplexSystem.impl`. The `system` implementation is used to aggregate two `process subcomponents`, i.e., `SenderProcess` and `ReceiverProcess` accompanied by their respective implementations, i.e., `SenderProcess.impl` and `ReceiverProcess.impl`. `SenderProcess.impl` contains the sending `thread` of type `Sender` whereas `ReceiverProcess.impl` contains two receiving `threads`, both of type `Receiver`

Each component is self-contained and only references other components. Given that `ComplexSystem.impl` is instantiated and consequently transformed into an instance model, all those references are resolved and the result is a tree-like structured model where `ComplexSystem.impl` contains both `process` implementations and those again contain their `thread subcomponents`. This means that all formerly implicit references are made explicit in the instance model. This explicit references have consequences for the `connections` defined in each component. Previously, loosely defined `connections` from a `subcomponent` to an outgoing `port`, e.g.: `con1` from `sender.dataOut` to `dataOut`, are now connected to other `connections` defined in the parent component, e.g., `con2` from `senderProc.dataOut` to `receiverProc.dataIn`. This way there are `connections` created that formerly did not exist. Those `connections` are called semantic connections. A semantic connection always goes from an ultimate source to an ultimate target and consists of all declared `connections` in between, e.g., `dataOut`, `con1`, `con2`, `con3`, `dataIn`. This means that at each `port` that defines multiple incoming/outgoing `connections` the number of semantic connections is multiplied by the number of incoming/outgoing `connections`. In Figure 2.9 this means we have two semantic connections:

- SemCon1: Sender.dataOut, con1, con2, con3, Receiver.dataIn
- SemCon2: Sender.dataOut, con1, con2, con4, Receiver.dataIn

Semantic connections are important for our approach as they are the actual `connections` over which data is transmitted. Depending on the `connections` they consist of, they change their behavior, their source or target or if they are active or not. The properties that influence the behavior of `connections` and thus, semantic connections are explained in detail in subsection 2.2.4.

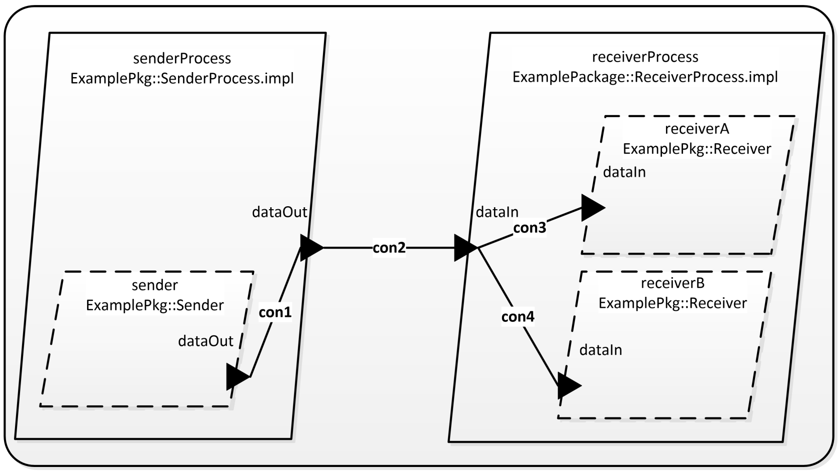

Figure 2.9.: Example system with two semantic connections.

2.2.4. Properties

Properties are used in AADL to either change the semantics of a model element or to add additional metadata to a model element. AADL comes with a variety of different properties that are preorganized into different `property sets`, e.g.: `Thread_Properties`, `Communication_Properties` or `Programming_Properties`. Properties can be declared by virtually every model element in AADL, be it a component type or implementation, a `feature` or a `connection`. As AADL predefines so many properties we restricted them to those important for our approach and categorize them according to their place of purpose, i.e., `threads`, `ports` or `connections`.

2.2.4.1. Thread Properties

The behavior of a `thread` can be altered depending on which values are given for several of its properties or properties of its `ports` or `connections`. The following represents a list of properties that can be used to alter the behavior of a `thread` directly in AADL:

- `Dispatch_Protocol`
- `Priority`
- `Period`
- `Deadline`

However, there are even more properties that (sometimes indirectly) have an effect on the runtime semantics of `threads`, like

- `Input_Time/Ouput_Time` on `ports` of the `thread` or
- `Timing` on `connections` between `threads` which we will investigate later in detail.

In the following we will describe each of those in detail.

Dispatch_Protocol A `thread` in AADL can have different types of dispatch:

- `periodic`
- `sporadic`
- `aperiodic`
- `background`
- `timed`

A `thread` is `periodic` if its code is executed recurringly with a given `period`. It is `sporadic` if there is no real `period`, but at least a minimum interarrival time which states how much time at minimum elapses before this `thread` is dispatched again. An `aperiodic` `thread` finally is completely event-triggered with no `period` or minimum interarrival time. `Background threads` are `threads` that consume left-over computing time and run (as the name implies) in the background. `Timed threads` are `threads` that are dispatched exactly once at a predefined time. This time can be relative to an event or absolute, e.g., at 05.09.2019 22:30:45.

Due to limitations of time, scope and the possibilities of RTSJ itself, we limit the possible values of `Dispatch_Protocol` to the value `periodic`.

Priority Most real-time systems use the concept of thread `priority` - usually an integer, where a higher value represents higher `priority` - in order to determine a feasible schedule for the given system. The `priority` is an indicator for the system of a desired order of execution for a given set of threads. Through a feasibility analysis a real-time system can determine if a given set of threads and their `priorities` have a feasible schedule which concedes each thread the computing time it desires. During runtime of such a real-time system a scheduler (in RTSJ a preemptive priority-based first-in-first-out scheduler [20]) then takes care of dispatching, stopping, and preempting threads according to the determined schedule.

Period Another variable of the aforementioned feasibility analysis is the `period` of periodically dispatched `threads`. The `period` determines when and how long at maximum this `thread` runs. Given a default of 1000 ms and 0 ns a `thread` runs each second for at most 1 second if no `deadline` (see below) is given. The `period` especially becomes important when it comes to the sampling of data exchanged between two `periodic threads` which is explained in detail in subsubsection 2.2.4.3.

Deadline As mentioned above, if no `deadline` is given, the `thread` can run at most for 1 second, as it then is restarted by its `period`. If one wants to restrict the maximum amount of time a `thread` may run, one should define a specific `deadline`. The `deadline` should be lesser or equal to the `period` if a `thread` is periodic, as it otherwise would still be allowed to compute when it is dispatched again. The `deadline` is another important variable for the feasibility analysis, as it allows restricting the maximum computation time of a periodic `thread` and thus providing the rest time of this `thread's period` as compute time for other `threads`.

2.2.4.2. Port Properties

Although there are many more properties, like `Queue_Size`, `Transmission_Type` or `Queue_Processing_Protocol` that affect the different types of `ports`, like `data ports`, `event ports` or `event data ports`, we decided to restrict those to the three most important for our approach. Those properties enable a minimal setup while we assume sensible defaults for others.

Input_Time / Output_Time `Input_Time` and `Output_Time` are both used to specify a time range, based on one of the lifecycle events of a `thread`, at which incoming `data` is frozen or outgoing `data` is being sent. Frozen means that the incoming `data` is made immutable for the time of the current dispatch cycle. Usually for `Input_Time` only the lifecycle events dispatch and start are sensible, while the lifecycle event plus the time range should not exceed the `deadline` of a given `thread`. Otherwise the `port` declares to receive `data` at a point in time where the corresponding `thread` can definitely not any further process this `data`. Vice versa this holds true for `Output_Time`, where the only sensible lifecycle events are completion and deadline and the lifecycle event minus the given time range should not be earlier than the start lifecycle event of the corresponding `thread`. Otherwise the `port` declares to send `data` that definitely cannot have been produced yet.

Fan_Out_Policy The `Fan_Out_Policy` determines how a `port` deals with `data` that is sent over it. AADL predefines four different possible values for `Fan_Out_Policy`. Due to the scope of this work and its corresponding reference implementation we consider `Broadcast` and `OnDemand` to be sufficient to deal with the majority of use-cases encountered in our test scenarios.

- `Broadcast`: `data` sent by this `port` is also sent over each `connection` connected to this `port`. Subsequent `ports` connected via these `connections` of course can have different `Fan_Out_Policies` so that in another part of a semantic connection the `data` might be cached.

- `OnDemand`: the need to explicitly request data from a data port which forces the port to store incoming data until the receiver explicitly demands it to be sent further.

2.2.4.3. Connection Timing

The only property of interest in our approach for `connections` is their `timing` property. This property interferes with those of `ports` and `threads` and is of uppermost importance for `periodic threads` that communicate via `data ports` and a `port connection`. The `timing` property has three possible values:

- `Sampled`
- `Immediate`
- `Delayed`

which will be explained in detail below.

Sampled A sampling `connection` between two `periodic threads` is the simplest case that can occur for `connections` between two `data ports`. In the upper half of Figure 2.10 we depicted two `periodic threads` with the same `period` that are communicating via a `port connection` between two `data ports`. Given this case, the receiver receives and processes the incoming `data` at its own pace of 10 times per second (period of 100ms).

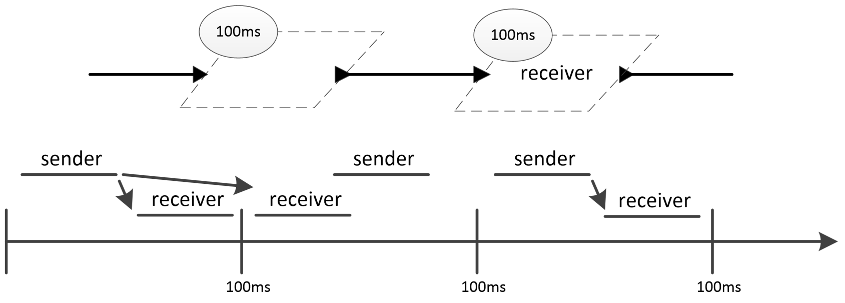

Figure 2.10.: Sampling port connection with over- and undersampling.

This means that the receiver does not pay any attention to the speed at which the sender produces `data`, neither does it regard the different start times within a dispatch cycle. This may lead to two different effects: over- or undersampling of `data`, which are both depicted in the lower half of Figure 2.10. In the case of oversampling, which can be seen in the first and second dispatch cycle, the receiver operates both times on the same `data`, as the sender has not been started in between the two times the receiver has been started. In contrast, the second and third dispatch cycle show the possible case of undersampling whereby the situation is vice versa. The sender is started twice before the receiver is started again. This results in lost `data` that is never received and thus not processed by the receiver.

Immediate One possibility to overcome the flaws of purely sampling `connections` is to declared a `connection` in AADL to be `immediate`. In Figure 2.11 we depicted an `immediate connection` (denoted via the parallel arrows on the `connection`) between two `threads` in the upper half of the figure. The lower half shows the effects of such a declaration. The receiver is forced to wait for the sender to produce and send its `data` before the receiver is dispatched and can receive and process that `data`. This way it is assured, that in each dispatch cycle the receiver receives new `data` and does not receive the `data` produced one dispatch cycle earlier. The functionality of waiting for another `thread` has to be implemented outside the responsibility of a scheduler and thus, has an impact on the implementation of `threads`. Also a drawback of this approach is the possibility of the receiver missing the end of a dispatch cycle although the compute time of both, sender and receiver, would easily fit. This can occur whenever the sender's actual start time is very late within a dispatch cycle and thus, delaying the start time of the receiver even further. A possible solution to this challenge is explained in the next section.

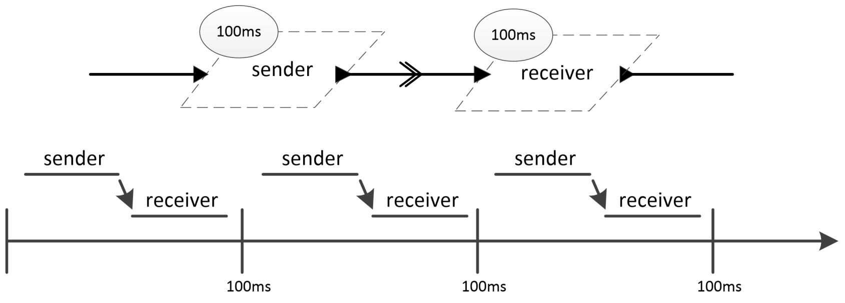

Figure 2.11.: Immediate port connection with deterministic sampling.

Delayed A countermeasure to the above mentioned challenge when using `immediate connections` is to declared the `connection` to be `delayed` instead. Figure 2.12 depicts in its upper half the declaration of a `delayed connection` between two `threads`, this time denoted via two parallel strokes. The lower half explains the gained effect of declaring a `connection` as `delayed`. The `data` produced by an arbitrary sender is delayed until the end of a dispatch cycle before it is transmitted to the respective receiver. This way the receiver deterministically receives new `data` in each dispatch cycle, even if the sender is running before the sender. As the receiver is not forced to wait within one dispatch cycle the risk of missing the end of a dispatch cycle is avoided by paying the price of receiving "old" `data`. `Delayed connections` have a direct effect on the source `ports` they are connected to, because the `Output_Time` of this `port` is overwritten.

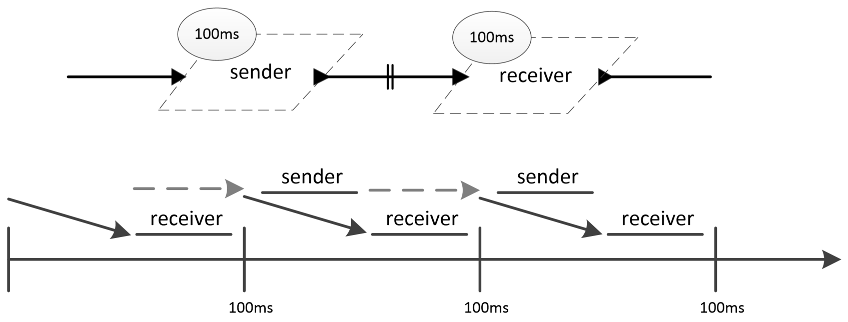

Figure 2.12.: Delayed port connection with deterministic sampling.

2.3. Real-Time Specification for Java

"The RTSJ is the specification resulting from JSR-1, the first specification launched through the Java® Community Process. [...] RTSJ is designed to support both hard and soft real-time applications. Among its major features are: scheduling properties suitable for real-time applications with provisions for periodic and sporadic tasks, support for deadlines and CPU time budgets, and tools to let tasks avoid garbage collection delays." [21]

Before RTSJ it was common wisdom for software developers that Java could not be used to develop real-time capable systems. Before we explain why this was true before RTSJ and why this fact has changed, we will give a short overview of what real-time systems are and what makes them stand apart from "normal" systems.

2.3.1. Real-Time Systems

Real-time systems are

"[...] systems that have to respond to externally generated input stimuli (including the passage of time) within a finite and specified time interval." [22]

Besides this definition [23] also identified the following characteristics:

Large and complex Complexity is rising. A statement that is true for many areas of our daily life. However, in this work we want to restrict its meaning to the area of real-time systems. Although, real-time systems can be very small, e.g. a controller for the rotor of a quadrocopter, they usually tend to grow very fast as soon as they are integrated into larger system, e.g., an autopilot system for a quadrocopter. A commonly used metric for complexity in software are SLOC. Staying in the domain of airborne software, which is a subset of real-time software systems, we see in Figure 1.1 the SLOC in millions for USAF aircrafts of the last three decades. The curve implies that SLOC double about every 4 years which implies a quadratic rise in complexity.

Extremely reliable and safe Real-time systems very often control some of society's most critical systems, e.g., nuclear power plants, aircrafts, railways or cars, just to name a few. Therefore, those systems must be written with uppermost safety and security in mind during development. Even in the unlikely case of a malfunction or failure, the overall system state has to be stable which is not always as easy as it seems, e.g., in case of a flying airplane. Even if failures occur the system shall be able to further operate, although, as the case may be, in a degraded way. A worst-case scenario still must be a **controlled** shutdown. [22]

Real-time control facilities

> *"Given adequate processing power, language and run-time support is required to enable the programmer to*
>
> - *specify times at which actions are to be performed*
> - *specify times at which actions are to be completed*
> - *respond to situations where all the timing requirements cannot be met*
> - *respond to situations where the timing requirements are changed dynamically."* [22]

Those four possibilities are called **real-time control facilities**. They enable a programmer to control succession of tasks, their timely start and stop or their frequency among others. Thereby, a programmer never directly manipulates the schedule of a system (which is usually determined by the system itself), but merely provides information to the system that enable it to determine a feasible schedule on its own.

Interaction with hardware interfaces Very often real-time systems are also embedded systems, which requires them to interact with the external world. Usual interactions happen through actuators, e.g., the rotors of a quadrocopter, and sensors, e.g., a barometer or GPS. Usually those devices get a predefined memory where they can write and read values from. Those values have to be in a defined format, e.g., a 32-bit Integer, unsigned, low endian. In order to enable a real-time system to interact with its environment the used programming language must provide means to access these memory areas, may it be through native provided functions or by providing interfaces to other languages that enable such access. [22]

Efficient implementation and a predictable execution environment Given the time-centric nature of real-time systems an efficient implementation will be more important than in usual systems. A telling example is the control software of an engine. Given an engine of a motorcycle, computation cycles with a few microseconds only are not uncommon. Some programming languages do not even provide clocks with such a fine granularity, neither any possibility to write code which runs in such short cycles and does not miss the deadline. Real-time systems not only have to run in such extremely short cycles they even have to do it in 100% of the time. They may not even once miss their deadline if they are defined as a hard real-time system which we will be explained in detail in the next section.

2.3.1.1. Soft, Hard and Isochronal Real-Time Systems

Real-time systems are often misunderstood in terms of which requirements must be fulfilled in order to be considered as a real-time system. Hard or soft real-time have neither to do with the size of the deadline nor with the speed the overall systems work at. A program reacting to an incoming event within two days can still be a real-time system, even a hard or isochronal real-time system which are the most restrictive ones. Usual systems are considered to be working correctly if they always produce the right answer to a given question. A real-time system additionally has to deliver this right answer within a given time (deadline). Depending on its classification - soft, hard or isochronal - it has to do so mostly, always or must always respond and even may not answer before another given time (earliest) elapsed. Hence, the three categories can be described as follows:

- **Soft real-time**: A system where the right answer has to be delivered in the majority of cases within a given deadline. If not the system is considered to be working in a degraded state of service.

- **Hard real-time**: A system where the right answer has to be delivered in every case within a given deadline. If not, the system is considered to be in an abnormal, non-functioning state.

- **Isochronal real-time**: A system where the right answer has to be delivered in every case within a given deadline and **not** before an earliest time has elapsed. If not, the system is considered to be in an abnormal, non-functioning state.

This way the aforementioned system responding within two days is still a hard real-time system if the given deadline is three days. It even might be an isochronal real-time system if an earliest time of one day is given. However, there might be cases where our fictional system needs four days instead of two and hence would be at the most considered a soft real-time system. Summarizing, real-time systems are systems that can be predicted in terms of time, be this time one nanosecond or five weeks. [24]

2.3.2. Latency and Jitter

Besides the definition of soft, hard and isochronal real-time, latency and jitter are the most important variables to consider when developing real-time systems. As with the different categories of real-time, latency and jitter are all about the overall predictability and determinism of a system. A real-time system must always behave in a way that can be predicted mathematically. This is where latency and jitter come into play.

> *"Latency is a measure of time between a particular event and a system's response to that event [...]"* [24]

In Figure 2.13, Latency in microseconds is depicted for a run of Cyclictest [25] on a normal Linux OS (red) and for a Linux OS with its real-time patch enabled (green). The average latency for normal Linux is 28 microseconds and for real-time Linux 27 microseconds. The other variable that can be read from this Figure is the jitter of the systems.

> *"Jitter is the variation or unsteadiness in a measured quantity."* [24]

Jitter in Figure 2.13 is represented by those samples that are far slower than the average latency of 27/28 microseconds. For the non-real-time Linux there are many samples in the area of 100 - 200 microseconds and even a few completely out of range with 350 microseconds and above. However, the Linux with real-time patch enabled shows a far smaller jitter with its highest latency being below 100 microseconds.

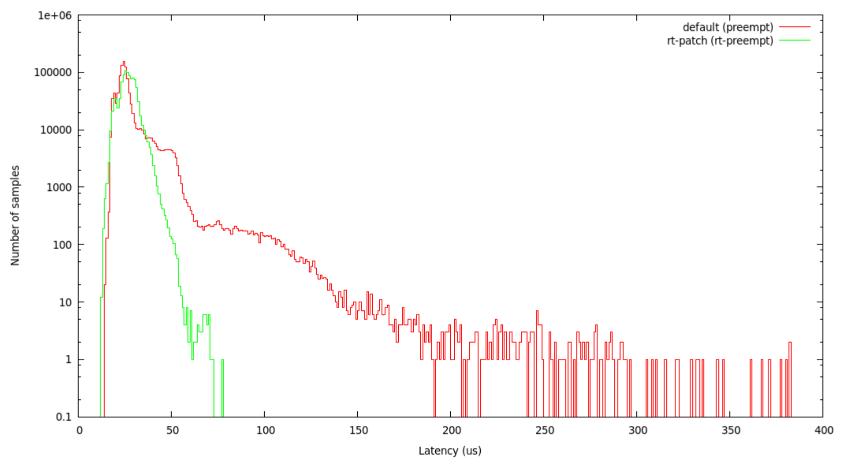

Figure 2.13.: Latency and Jitter for Linux with and without rt-patch. [26]

2.3.3. Standard Java

After having seen what makes a real-time system stand apart from "normal" systems we will now have a closer look on what makes normal Java incapable of being a programming language for real-time systems and how the RTSJ changes this.

2.3.3.1. Garbage Collection

Garbage Collection is what made Java once stand apart from other programming languages where programmers were forced to take care of memory allocation and deallocation by themselves. Java does not provide features to explicitly allocate or deallocate memory, but works with objects for which the JVM itself allocates memory and deallocates it, if the given object is freed by the garbage collector. In Java there are currently 5 different Garbage Collectors:

- **Serial Collector**: single threaded, pauses the whole application when performing a collection

- **Parallel Scavenging Collector**: only works for the young generation of objects, older generations are collected by the serial collector
- **Parallel-Compacting Collector**: similar to Parallel Scavenging Collector, but also working for old generations
- **Concurrent Mark-Sweep Collector**: improvement for above named collectors, significantly reduces the pause during collection
- **G1 Garbage Collector**: successor of the Concurrent Mark-Sweep Collector, minimizes the pauses due to garbage collection even further

Common to all collectors is the pause they cause, which is neither predictable in its length nor when it occurs. This sort of unpredictability can render the best WCET prediction void and thus makes normal Java an inappropriate choice for writing real-time systems.

2.3.3.2. Just-In-Time HotSpot Compilation

Another real-time deficiency of Java is the Just In Time (JIT) HotSpot compiler. Often Java's bytecode is initially only interpreted by the JVM and therefore rather slow in execution time. This is where the JIT HotSpot compiler comes into play. Given the case that a method is called more than 10000 times, which is the default, the JIT compiler first translates the bytecode of this method into machine code which can be directly run on the processor instead by the interpreter of the JVM. This is only the first stage of the JIT compiler which also recompiles already compiled code into even more efficient machine code if the same method is called more often. On the one hand this mechanism leads to a so called warm-up phase of the JVM in which code is executed significantly slower than in later phases. On the other hand those relatively arbitrary, additional compilation phases render the whole program unpredictable in terms of WCETs. We do neither know when the compilation of methods takes places, neither do we know the time it takes to (re-)compile these methods into machine code. Again, this sort of unpredictability makes normal Java an inappropriate language for writing real-time systems.

2.3.3.3. Thread Priorities

After two sources for unbounded latency and jitter we will now have a look on how Java handles threads and their priorities. In subsection 2.3.1 we worked out real-time control facilities to be one characteristic of real-time systems. Usually this means features that enable programmers to synchronize their code with time itself via precise clocks, periodic event sources or language features encapsulating time in an absolute or relative form as well as using those to determine the start and end of tasks. Another mandatory feature is to enable programmers to specify the importance of a task or in our case a thread. Usually this is done via priorities which Java indeed offers. Unfortunately, most JVM implementations completely ignore the priority values given to a specific thread. This implies that despite having a priority of 10 a thread would be granted the same computation time as another thread with priority one. Far worse, the original Java Language Specification stated the following:

> *"Every thread has a priority. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to implement mutual exclusion."* [27]

Due to this statement Java cannot guarantee thread priority obedience and without this capability we are unable to build real-time applications. This means that even if we were able to control the garbage collector and the JIT compiler in a predictable way, we are still unable to guarantee real-time behavior for our application. [24]

2.3.3.4. Hardware Access

As stated in subsection 2.3.1, real-time systems usually need access to their surrounding environment. This might be sensors and actuators that write and read data to/from predefined memory areas for control and data delivery. Because of its platform independent nature Java grants no direct access to the underlying memory instead providing its own memory model. Reading from or writing to sensors and actuators is therefore completely impossible with Java's language features alone. One possibility to circumvent this is to access functions written in other languages via Java Native Interface (JNI), but this interferes with Java's

"Write Once Run Anywhere" goal. Although not a complete show stopper this missing feature definitely makes Java even less attractive for writing real-time applications.

2.3.4. Real-Time Java

RTSJ is the first Java Specification Request (JSR) ever launched. This indicates the enormous importance Java developers ascribe to real-time capabilities for Java. As part of JSR-1 all of the above named deficiencies of Java regarding writing real-time applications are addressed and solved as we will see in detail in the following subsections.

2.3.4.1. Real-Time Garbage Collection

Recall the example of an application running for days and still being a real-time application, as long as it is predictable and does not miss its deadlines. What holds true for this application also does for a garbage collector. Real-time garbage collectors are not about being fast, but about being predictable (while still being reasonable fast). Two common approaches to achieve such garbage collection behavior are work-based garbage collection and time-based garbage collection. As our targeted JVM implementation, JamaicaVM [15], uses a work-based garbage collector we will explain its details in the following.

Work-Based Garbage Collector A work-based garbage collector becomes predictable by ensuring that every time a thread, no matter what priority, allocates an object, it must pay some of the cost of garbage collection work up front. Usually the garbage collection work does not vary much between allocations, thus making it predictable in terms of WCET for the thread, e.g., the manual for the JamaicaVM shows an example build of a common "HelloWorld" application where a memory analyzer tool is used to measure the application's memory usage (heap). The analysis' output consists of the maximum heap memory demand plus a table of possible heap sizes and corresponding worst-case allocation overheads, which are given in units of garbage collection work that are needed to allocate one block of memory (typically 32 bytes). The real amount of time is platform dependent, e.g., on the PowerPC processor a unit corresponds to the execution of about 160 machine instructions. In their example the application uses a maximum of 3362144 bytes of memory for the java heap, the heap sizes are given from 3732K to 17373K and the constant garbage collection work from 3 to 100 units. Garbage collection work and heap sizes are inversely proportional to each other, i.e., a larger heap means fewer garbage collection units. This way a programmer is able to determine exactly how much time he must allow for the WCET of a given thread. [28]

2.3.4.2. Ahead Of Time Compilation and JarAccelerator

As explained in subsubsection 2.3.3.2, Oracle's JVM employs a JIT compiler which (re-)compiles code during runtime. Another class of compilers are so called Ahead Of Time (AOT) compilers which are performing compilation during compile time instead of runtime. Although, JIT compilers in Java are usually faster, AOT compilers are a perfect match for real-time applications due to their predictability. All of the code is translated into byte code or machine code during compile time and no additional work has to be done during runtime, ergo no additional time is spent on (re-)compilation of source code.

The performance loss is still a drawback worth mentioning, as it has to be considered during development of real-time applications. While investigating our own code executed on the JamaicaVM we usually observed a slowdown by the factor of 10 compared to the execution time on a current Oracle JVM. These measurements are only meant to be taken as a very coarse indication, as there are many factors that influence the execution speed of Java programs. A possibility to speed up a real-time application with AOT compilation is to translate the source code not just into byte code, but directly into machine code of the target platform as it is partially done by the JIT compiler of Oracle's HotSpot JVM. This usually can only be done for an existing application. If any code has to be dynamically loaded afterwards one could use, although it is a vendor specific solution, Aicas' JarAccelerator which allows to AOT compile individual .jars to cpu machine instructions for dynamic loading at runtime.

Using AOT compilers and tools like Aicas' JarAccelerator enable programmers to reliably predict execution times of their application, thus making them real-time capable.

2.3.4.3. Real-Time Threads and Priority-Based Preemptive Scheduler

As stated in subsubsection 2.3.3.3, usual JVMs do not care about priorities of threads, let alone have a scheduler being able to preempt threads based on their priorities. By contrast, RTSJ requires an implementation to provide (at least) a fixed-priority preemptive scheduler with at least 28 unique priority levels [20]. The JamaicaVM provides an implementation which is a first-in-first-out fixed-priority preemptive scheduler. This scheduler is able to create a feasible schedule for a set of given `Schedulables`, which is an additional `interface` defined by the RTSJ. This `interface` is initially implemented by two `classes` in the RTSJ, `RealtimeThread` and `AsyncBaseEventHandler` along with their subclasses.

Real-Time Threads `RealtimeThreads` are one of the core concepts of RTSJ and extend normal Java `Threads` as can be seen in Figure 2.14. The above mentioned `Schedulable` `interface` extends Java's `Runable` `interface`, hence making all RTSJ specific `classes` compatible with existing Java applications. The only difference entails a real-time JVM to treat `RealtimeThread` different from standard Java `Threads` according to their given `ReleaseParameters`, `SchedulingParameters` and `MemoryParameters`. The most common form of `SchedulingParameters` are `PriorityParameters` which indicate the importance of one thread towards another. `ReleaseParameters` enable a programmer to synchronize its code with time itself through providing methods to set a deadline or in case of `PeriodicParameters` a period or start time. This additional information is taken into account by a scheduler's attempt to create a feasible schedule.

Although a good starting point for programmers to write real-time capable applications, writing `RealtimeThreads` is more complicated than initially apparent, e.g., in order to create a periodic `RealtimeThread` we have to adapt its run method as depicted in Listing 2.5.

```java
public class MyRTThread extends RealtimeThread{

    @Override
    public void run() {
       while (Thread.interrupted()) {
           doSomething();
           waitForNextPeriod();
       }
    }
}
```

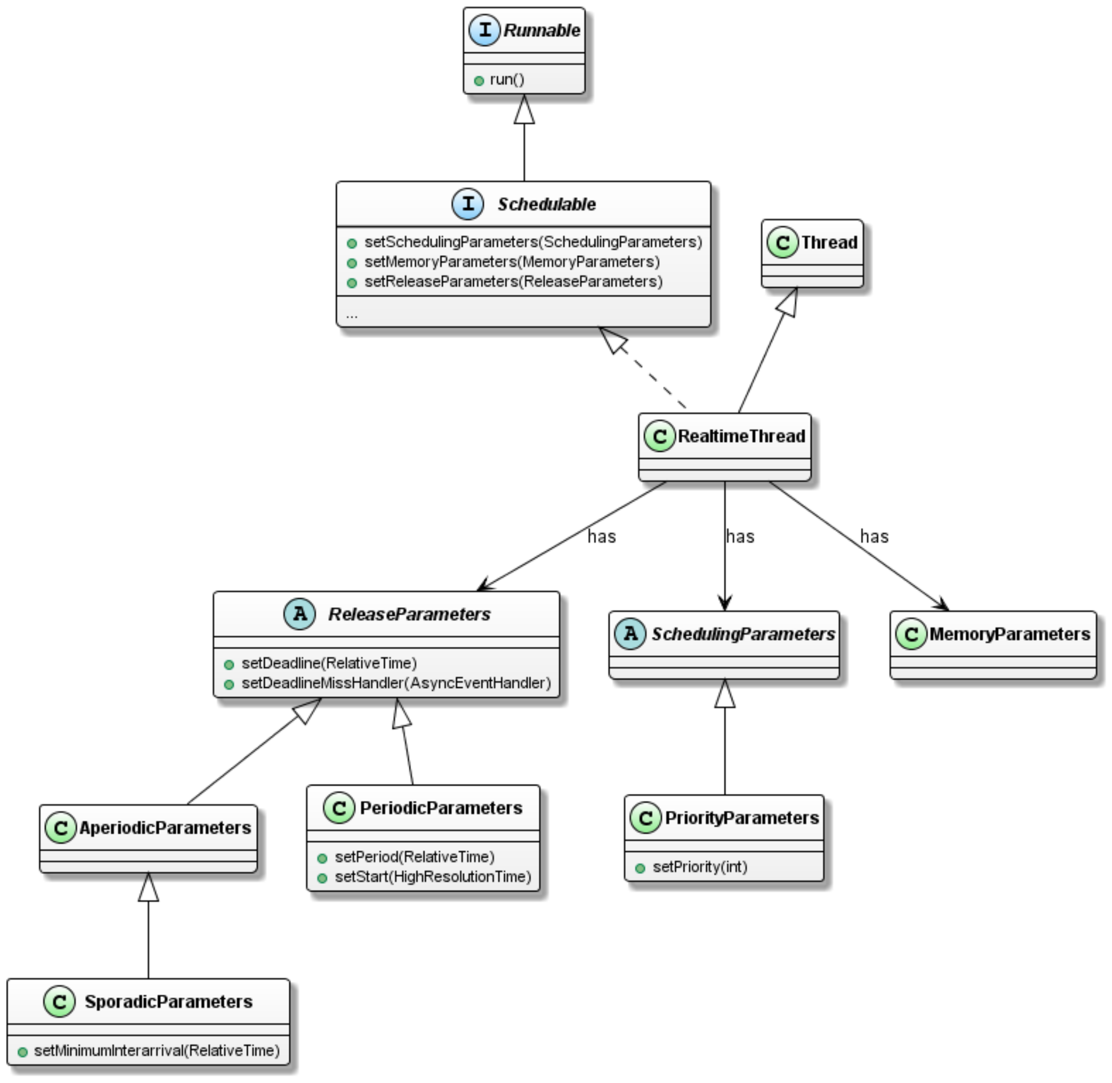
Runnable
run()
Schedulable
setSchedulingParameters(SchedulingParameters)
setMemoryParameters(MemoryParameters)
setReleaseParameters(ReleaseParameters)
...
Thread
RealtimeThread
has
has
has
ReleaseParameters
setDeadline(RelativeTime)
setDeadlineMissHandler(AsyncEventHandler)
SchedulingParameters
MemoryParameters
AperiodicParameters
PeriodicParameters
setPeriod(RelativeTime)
setStart(HighResolutionTime)
PriorityParameters
setPriority(int)
SporadicParameters
setMinimumInterarrival(RelativeTime)

Figure 2.14.: RTSJ `RealtimeThread` and its associated classes and interfaces.

Listing 2.5: RTSJ `RealtimeThread` with `PeriodicParameters` and accordingly adapted `run` method.

A naive expectation would have been that the JVM takes care of calling the `run` method periodically and putting the `RealtimeThread` to sleep after its `run` method has been executed. Instead the programmer still has to take care of forcing the `RealtimeThread` to run periodically, e.g., by using a `while` loop, and to give away its spare time, if any, by calling `waitForNextPeriod()` which lets this

RealtimeThread wait as long as it takes to reach the end of its period given by its PeriodicParameters. As this approach is rather unintuitive, we will present another approach in the next section.

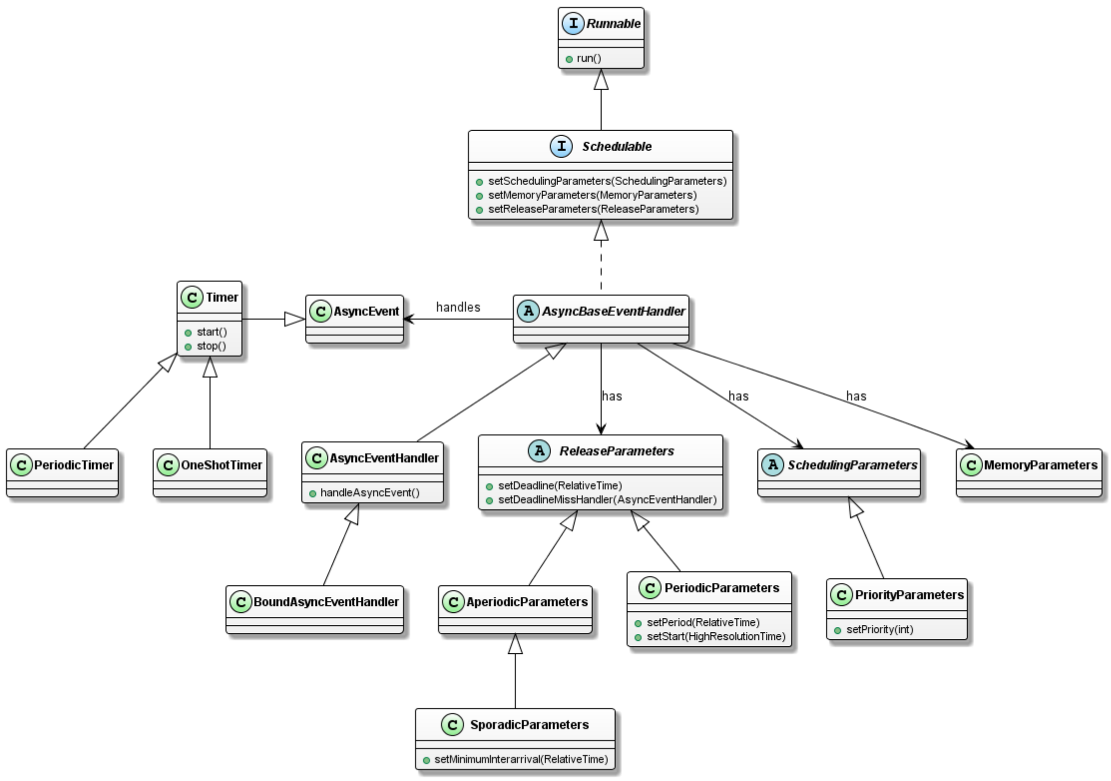

Figure 2.15.: RTSJ `AsyncBaseEventHandler` with subclasses and associated classes and interfaces.

Asynchronous Event Handlers Although initially merely intended for asynchronous events, AsynchronousEventHandlers can also be used to write real-time applications without direct use of RealtimeThreads. As can be seen in Figure 2.15 the base class of all AsynchronousEventHandlers is AsyncBaseEventHandler. Like RealtimeThread it implements Schedulable but in contrast to RealtimeThread it does not directly extend Java's Thread. Instead it has an association to an AsyncEvent which can be a representation for a system event or an interrupt or simply a timer, e.g., in order to use an AsyncEventHandler for periodic tasks we have to add it to the list of AsyncEventHandlers of a PeriodicTimer as shown in Listing 2.6. ReleaseParameters and SchedulingParameters are set the same way they would be set for a RealtimeThread. In the background the JVM takes care of associating the AsyncEventHandler with an appropriate `RealtimeThread` regarding the given `SchedulingParamters` and `ReleaseParameters`. All a programmer is left with is to write the business logic of its `AsynchronousEventHandler`. Thus, the business code is kept free from timing related code.

```java
PeriodicTimer timer = new PeriodicTimer(null, new RelativeTime(100, 0), null);
AsyncEventHandler handler = new AsyncEventHandler(){

    @Override
    public void handleAsyncEvent(){
        doSomething();
    }
}

ReleaseParameters rps = timer.createReleaseParameters();
timer.setSchedulingParameters(new PriorityParameters(11));
rps.setDeadline(new RelativeTime(50, 0));
handler.setReleaseParameters(rps);

timer.addHandler(handler);
timer.start();
```

Listing 2.6: `PeriodicTimer` and `AsyncEventHandler`.

Bound vs. Unbound Asynchronous Event Handlers The mentioned benefit of using `AsyncEventHandler` over `RealtimeThread` can also be a drawback, if a programmer does not exercise care when using it. A common pitfall is to use `AsyncEventHandler` when waiting for other tasks to finish. Recall, the JVM takes care of associating an `AsyncEventHandler` with an appropriate `RealtimeThread` in the background. Now, imagine the case where a programmer has two `AsyncEventHandlers`, a producer and a consumer. The consumer waits for the producer to finish, e.g., calls `wait()` on a shared resource and waits for a `notify()`. If both, producer and consumer, have the same priority the JVM might decide to associate them with the same `RealtimeThread` in the background. In case the consumer is running before the producer, which is completely legitimate, and then calls `wait()` on the shared resource it will stop the thread from working. Consequently, the producer will never run (it should be run by the same `RealtimeThread`) and will never call `notify()` on the shared resource. The result of such a scenario behaves like a deadlock without the explicit use of locks.

A solution for this situation is the use of a `BoundAsyncEventHandler`. A `BoundAsyncEventHandler` is guaranteed to be associated with its own, unique `RealtimeThread`. Consequently, a situation as described above could never happen as the JVM will always associate two `AsynchronousEventHandler` with two different `RealtimeThreads`. It also would be sufficient to only make the consumer be a `BoundAsyncEventHandler` as it is the one that is blocking the thread.

2.3.4.4. Hardware Access

As mentioned in subsection 2.3.1 real-time systems often need access to their surrounding environment. This is usually done via devices that have their interface registers mapped into the virtual memory address space. RTSJ provides two mechanisms for granting access to this memory:

- **Direct Memory Access/Shared Memory**: *"Mechanisms that allow objects to be placed into areas of memory that have particular properties or access requirements."* [24]
- **Raw Memory Access**: *"Mechanisms that allow the programmer to access raw memory locations that are being used to interface to the outside world, e.g., memory-mapped input or output devices."* [24]

```java
public class MemoryAccess {
  private static final long BASE_ADDRESS = 0x3f000001;
  private static final long GPIO_OFFSET = 0x2000001;
  private static final int LENGTH = 32;

  public static void main(String[] args) {
    final RawInt mem = RawMemoryFactory
            .getDefaultFactory()
            .createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
        BASE_ADDRESS + GPIO_OFFSET, LENGTH, 1);
    }
}
```

Listing 2.7: Raw memory access via RTSJ.

Listing 2.7 shows an exemplary access to a `RawInt` that is stored in a specific place in memory. In order to access this specific place we create the default instance of `RawMemoryFactory` on which we call the method `createRawInt` which takes parameters according to [29] as follows:

- ***region***: *"The address space from which the new instance should be taken."*
- ***base***: *"The starting physical address accessible through the returned instance."*

- ***count***: *"The number of memory elements accessible through the returned instance."*

- ***stride***: *"The distance to the next element as a multiple of element size, where 1 means the elements are adjacent in memory."*

In our case we want to access a memory-mapped region of a Light Emitting Diod (LED), hence we are using the predefined `RawMemoryRegion` `MEMORY_MAPPED_REGION`. The `base` is calculated via a `BASE_ADDRESS` plus a `GPIO_OFFSET`, `count` is given via the `LENGTH` variable (a 32-bit integer) and `stride` is 1. We are now able to switch the LED on and off by writing 0, respectively 1 to the memory via `mem.setInt(value)`.

As shown by this example RTSJ gracefully solves the challenge of Java's non-existent hardware access by encapsulating hardware access into a facade, i.e., `RawMemoryFactory`, and by doing so delegating the implementation details to the real-time JVM vendor. Additionally, `RawMemoryFactory` enables a type-safe access to the underlying memory which eliminates a possible source of errors. Also the challenges of priority-based preemptive scheduling as well as real priorities have been tackled and solved by the RTSJ as explained in subsubsection 2.3.3.3. The standard even forces compliant implementations to offer a real-time garbage collector which erases the need for manual memory management. AOT compiler and proprietary tools like Aicas' JarAccelerator offer a sensible trade-off between runtime performance and predictability.

Based on the demonstrated fact of Java being suitable for writing real-time applications and its undisputed supremacy in the rambling market of programming languages [10], we chose Java as starting point for the technologies and approaches shown later in this work.

2.4. Open Services Gateway initiative

OSGi has been around since the year 2000, where its first specification was released. According to the official website of the OSGi Alliance OSGi is defined as follows:

> *"OSGi is a set of specifications that define a dynamic component system for Java. These specifications enable a development model, where applications are dynamically composed of many different reusable components. The OSGi specifications enable components to hide their implementations from other components while communicating through services, which are objects that are specifically shared between components. This architecture significantly reduces the overall complexity of building, maintaining and deploying applications."* [30]

As we decided to use Java as programming language throughout this work, we will explain OSGi as it is implemented in Java, although other implementations in other languages exist [31]. OSGi is a dynamic module system on top of Java that offers an approach for several shortcomings and common pitfalls that occur when developing large systems in Java. To provide a better understanding of what OSGi exactly is and how it compares to a normal Java application we depicted an abstract view of an application written with OSGi and an application written in plain Java in Figure 2.16.

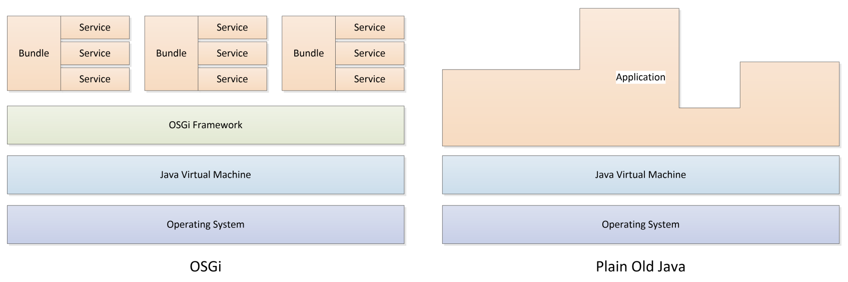

Figure 2.16.: OSGi application vs. plain old Java application.

On the right side we can see a normal Java application which is more or less a monolithic application that runs on the JVM which in turn runs on an Operating System (OS). Somewhere within this Java application there is a `main` method that is invoked by the JVM at startup and the rest of the application's lifecycle is up to the developer.

On the left side we can see an application written with OSGi. The main difference is shown by the green box which represents the OSGi framework. On top of this framework a developer creates so called **bundles** which are Java .jar files with additional information in their MANIFEST.MF file. Each bundle contains one or more **services** that are normal Java `classes` with well-defined `interfaces` that contain program logic. Where the Java application has been more or less monolithic in its nature, the OSGi application is modular by default. Each service can only be accessed through its `interface` and each bundle can decide which `interfaces` and `classes` it makes accessible to other bundles. Also each bundle and service has their own well-defined lifecycle that is controlled by the OSGi framework and that can be hooked into by a developer. Hence, the `main` method is hidden somewhere in the OSGi framework and not created by the developer.

In the following we will have a more detailed look at why we use OSGi, how bundles and services work in detail as well as the runtime reconfiguration capabilities of OSGi services.

2.4.1. Reasons for OSGi

As SCSS tend to follow the overall trend of software in growing fast, OSGi is a reasonable solution to counter this trend when already using Java. The usage of OSGi comes with a set of benefits according to the OSGi Alliance [32]:

Reduced Complexity When developing with OSGi one develops bundles. Bundles are simple .jar files with additional information in their MANIFEST.MF file that will be explained in subsection 2.4.2 in detail. Each bundle has a set of dependencies to other bundles (requirements) as well it offers a set of `interfaces` and `classes` to other bundles (capabilities). Usually, each bundle only represents one well-defined functionality (often only one service with its `interface`) and therefore is rather small in terms of number of `classes` and lines of code. This makes them easy to understand and better maintainable than monolithic code where dependencies are not clearly defined. More complex applications are achieved by composing several of these small bundles to achieve another, aggregated functionality. An example for this might be a booking system, where each step of the booking (find offers, submit booking request, check credit card information, ...) is represented by a small bundle and the whole booking application is just an aggregation and orchestration of those bundles and their services.

Reuse Following the aforementioned example, we can reuse each bundle in a different scenario. This way we might reuse the "check-credit-card-information" bundle within a cash system. The functionality of the single bundle stays the same, but it is used within a complete different scenario.

Dynamic Updates Bundles and services have their own lifecycles. Both will be explained in detail in subsection 2.4.2 and subsection 2.4.3, but we will already use some of their possible states here. Bundles can get **installed** and **uninstalled** at runtime which means that with no downtime a production server can be updated with the newest bugfixes. Also services can be **activated** and **deactivated** while their corresponding bundle is active. If we imagine an embedded application where a service represents a sensor, this lifecycle can be utilized to notify the overall system of the failure of this sensor as the OSGi framework handles newly created and vanishing bundles/services during runtime and broadcasts these changes to the rest of the system.

2.4.2. Bundles

As mentioned above the main building blocks of OSGi applications are bundles. In Java, bundles are represented as .jar files. A bundle represents a self-contained, cohesive module, often with a public API and a private implementation for this API. OSGi defines a dedicated layer in its architecture overview [30] for the import and export of code from and to bundles, whereby the API of a bundle is crucial as the API is the part of a bundle that is usually exported and subsequently imported by another bundle. Those import-export dependencies between bundles are used for dependency resolution during buildtime. Each bundle has its own classloader which differs strongly from Java's usual "one classloader for everything" philosophy. This makes it possible to have different versions of the same library within different bundles. If there were only one classloader this would not be possible as the names of `classes` within the different versions of a library would not be distinguishable.

Lifecycle All bundles have a predefined lifecycle as shown in Figure 2.17. Bundles can be `Installed`, `Resolved` and `Uninstalled` during the runtime of an application. After a bundle is installed, the OSGi framework resolves its declared dependencies to other bundles and, if all dependencies can be satisfied, sets its state to `Resolved`. Every bundle can also declare an `Activator` to participate in the lifecycle events after being set to `Resolved` and getting started and before being stopped and set back to `Resolved`. The `Activator` is a `class` whose methods are called whenever the bundle is started or stopped. In Figure 2.17 this is depicted by the states `Starting`, `Active` and `Stopping`. A bundle can either be started/stopped by the framework itself, e.g., whenever it gets resolved it is also started, or can be started/stopped manually by a user.

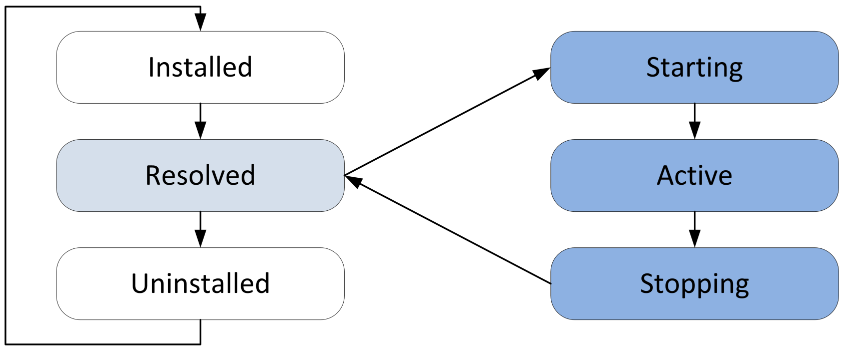

Figure 2.17.: States of a bundle's lifecycle [33]

Dependencies
Recall that bundles are merely .jar files which are zip containers containing all .java, .class, .jar and other files needed by this .jar to be complete. The .java files contain the Java source code written by a programmer and usually have an `import` section, where all `classes` are declared that neither are provided by the Java Runtime Environment (JRE) directly nor by the `package` the `class` is declared within. Those are imports that have to be satisfied by another bundle at runtime and are therefore dependencies to another bundle. This is why bundle dependencies are declared in form of `import` and `export` statements that declare the `packages` to be imported from another bundle or to be exported in order to be used by another bundle. Usually, all packages are also declared with a version range, e.g., de.unia.smds.app.api [1.0.0 - 2.0.0]. The versioning further restricts the packages that are allowed to satisfy a given dependency. Only when a bundle can be found that exports a matching package in the right version our bundle gets resolved.

2.4.3. Services

Although bundles pose a good possibility to modularize and organize code, they are not sufficient to capture all needs of modern software development. Therefore, a service model has been introduced by OSGi.

Services are the core of modern OSGi. In order to give a gentle introduction to OSGi services, we draw a direct comparison between a simple `Hello World` application written once with OSGi and once with plain Java. Both are depicted in Figure 2.18. Although, the OSGi version might look a little more verbose with its two annotations, it is still the same amount of characters. In plain Java we must always implement a `public static void main(String[] args)` method in order to offer an entrypoint for the JVM, whereas in OSGi we can annotate an arbitrary method with `@Activate`. Both code snippets provide the same output to the console.

```java
@Component
public class HelloWorld{
  @Activate
  private void hi(){
    System.out.println("Hello,
        OSGi!")
  }
}
```

```java
public class HelloWorld{
  public static void main(String[]
       args){
    System.out.println("Hello Java
        !");
  }
}
```

Figure 2.18.: Hello World! in OSGi on the left vs. in plain Java on the right.

The `@Component` annotation used in the OSGi snippet declares the annotated `class` to be an OSGi service. Those so called Declarative Service (DS) are the current state-of-the-art when developing OSGi applications.

2.4.3.1. Declarative Services

When OSGi once started in 2000, declaring services involved a lot of boilerplate code, making it a rather messy framework for beginners. Business logic often was tightly entangled with logic concerning the declaration, registration and lookup of services which lead to code that could not be used outside an OSGi framework. Also the dynamism of OSGi often posed a common pitfall to beginners as services that were once looked up still could vanish later on and, if not handled correctly, could lead to unexpected behavior or failure of the developed system.

DSs are a non-programmatical approach to declare services in OSGi which enable a programmer to write plain Java `classes` that are enriched with metadata via annotations. By using annotations the written Java `classes` are freed from most dependencies to an OSGi framework. The only dependency left is the one to the annotation package which only consists of a small number of annotations and enums [34]. This way a developer can write services without loosing the reusability of the written `classes` if they are repurposed in another, non-OSGi context. The above mentioned annotations will be explained in detail in the following paragraphs.

@Component The most important annotation is `@Component`. It declares the annotated `class` to be a component that is registered with its respective service(s), i.e., `interface(s)`, at the service registry. The `@Component` annotation comes with a set of predefined properties which are used to configure the behavior of the created component and can add additional metadata. We will shortly explain the most important ones for our later approaches according to [35]:

- `enabled`: A `boolean` indicating the initial state of this component when the containing bundle is started.
- `immediate`: A `boolean` indicating whether or not the annotated component is subject to immediate or delayed activation. Activation and component lifecycles will be discussed in the following.
- `name`: A `String` used as name for this component which must be unique within a bundle. Can be used by other components to filter for this exact component.
- `property`: An `Array of Strings` that specifies a set of component properties. Is used to add additional metadata to the service which can then be utilized by other components
- `service`: An `Array of Class<?>` that specifies the name(s) of the `interface(s)` or `class(es)` this component is registered under as a service.

Although the `@Component` annotation uses sensible defaults for all its properties and we are able to use it without explicitly setting any property at all, see Figure 2.18, in Listing 2.8 we exemplarily show the above named properties when being set explicitly. This example shows a simple version of a `Servlet` that can be found by a running `HttpService` and subsequently be published according to the properties set. So getting a `Servlet` up and running comes down to using the `@Component` annotation and adding additional metadata, i.e., the path the `Servlet` is registered under. The developer is only left with implementing the business logic.

```
@Component{
    enabled=true,
    immediate=true,
    service = Servlet.class,
    name="MyComponent",
    property= "osgi.http.whiteboard.servlet.pattern=/hello"
}
public class HelloWorldServlet extends javax.servlet.http.HttpServlet {
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException {
    resp.getWriter().println("Hello World");
  }
}
```

Listing 2.8: Usage of properties withing `@Component` annotation.

Within this component a developer can also use all the other amenities OSGi has to offer, e.g., lifecycle event methods.

@Activate, @Deactivate and @Modified Components live within a bundle and its lifecycle, so all component-specific lifecycle events take place within the `Active` state of its according bundle as depicted in Figure 2.17. Although there exist different forms of lifecycles for components which will be shown in subsubsection 2.4.3.2, every single component in OSGi can at most participate in three lifecycle events: **Activation**, **Modification** and **Deactivation**. To participate in these events a component must declare methods annotated with the corresponding annotation as depicted in Listing 2.9. Any method annotated with `@Activate` is called by the OSGi framework as soon as the declaring component is instantiated and started. The method can also declare a set of predefined parameters in order to offer more context information to the developer. Those parameters can be:

- `BundleContext`: Offers information about installed bundles and means to start/stop these or to search their classpath for resources or .class files.

- `ComponentContext`: Offers information about the component and offers means to modify the component itself.
- `ComponentPropertyType`: An object containing configuration information of this component.

While being activated a component can be modified if it contains a method annotated with `@Modified`. The possible parameters are the same as with `@Activate`, but usually the most interesting parameters in `@Modified` methods are `ComponentPropertyTypes`. Those enable a component to be reconfigurable at runtime which will be shown in detail in subsection 2.4.4. Whenever the component is deactivated, e.g., because its declaring bundle is stopped, the method annotated with `@Deactivate` is called. This method usually is used to clean up a component, e.g., shut down network connections or free resources obtained by this component.

```java
@Component
public class LifecycleShowcase {
    @Activate
    private void activate(){ System.out.println("Hello!"); }

    @Modified
    private void modified(){ System.out.println("Modified!"); }

    @Deactivate
    private void deactivate(){ System.out.println("Goodbye"); }
}
```

Listing 2.9: Component lifecycle methods via annotations.

`@Activate` and `@Deactivate` are also both interconnected with the `@Reference` annotation. `@Activate` is called after all references have been resolved and `@Deactivate` is called just before those references are freed again. The details on how `@Reference` works are shown in the next part.

@Reference Until now only standalone components have been shown, however `@Reference` can be used to link different components together. In Listing 2.10 we depicted a simple reference between a `HelloCommand` component and a `Greeter` service which is used by the `HelloCommand` component in its `@Activate` method. The reference is realized via a member variable of type `Greeter` within the `HelloCommand` component that is annotated with `@Reference`. The `interface Greeter` is implemented by `GreeterImpl` which is a component as denoted by the `@Component` annotation as is `HelloCommand`. The OSGi framework takes care of instantiating both components and injecting a reference into the member variable of `HelloCommand`. In the `@Activate` method this reference is already set and can thus be used to invoke the method `greet()` on it.

```java
@Component
public class HelloCommand {
  @Reference
  private Greeter greeter;

    @Activate
    private void activate(){ greeter.greet(); }
}

@Component
public class GreeterImpl implements Greeter{
  @Override
    public void greet(){ System.out.println("Hello World!"); }
}
```

Listing 2.10: Service reference via annotation.

Similar to `@Component`, `@Reference` can be modified in its behavior through properties. Taken from [35] the most important ones for our approach are the following:

- `cardinality`: An enumeration value describing if this reference has an `OPTIONAL` (0..1), a `MANDATORY` (1..1), a `MULTIPLE` (0..N) or an `AT_LEAST_ONE` (1..N) cardinality.

- `policy`: An enumeration value that can either be `STATIC` or `DYNAMIC`. `STATIC` forces the declaring component to restart whenever this reference changes, whereas `DYNAMIC` enables the declaring component to stay activated but still receive a new reference.

- `policyOption`: An enumeration value that can either be `RELUCTANT` or `GREEDY`. `GREEDY` leads this reference to be changed to another service whenever a better fitting one is available, whereas `RELUCTANT` is satisfied with the first reference it gets and does not change afterwards.

- `target`: A Lightweight Directory Access Protocol (LDAP) [36] filter expression as `String` which describes the to target service(s) via metadata that must be present at the target component.

Although `@Reference` uses sensible defaults for all its properties and we are able to use it without explicitly setting any property at all, see Listing 2.10, in

Listing 2.11 we exemplarily show the above named properties when being set explicitly.

```java
@Component
public class HelloCommand {
  @Reference(
      cardinality = ReferenceCardinality.MANDATORY,
        policy = ReferencePolicy.DYNAMIC,
        policyOption = ReferencePolicyOption.RELUCTANT,
        target = "(serviceRanking<=10)"
    )
    private Greeter greeter
}
```

Listing 2.11: Usage of properties within `@Reference` annotation.

In this example the `HelloCommand` component's `greeter` reference would not accept any `Greeter` service with a `serviceRanking` higher or equal than 10. This means that even if there is a component registered under the `Greeter` `interface` but has a `serviceRanking` of 11, the `HelloCommand` component would not accept a reference to it and thus not get activated as `greeter` is a mandatory reference. The possibility to change the target filter at runtime as later explained in subsection 2.4.4 enables the dynamic reconfiguration of components at runtime, which is a powerful mechanism in OSGi in order to create runtime reconfigurable systems.

2.4.3.2. Component Lifecycles

In subsubsection 2.4.3.1 we already had a look at lifecycle methods that are available for all DSs. Although these methods are the same for all DSs, their lifecycle might be different. Usually we differ between two types of DSs depending on whether or not the DS is immediate. We do this because depending on being immediate or not the lifecyle of the component has an additional state as depicted in Figure 2.19.

On the left side an immediate DS is depicted. A DS is either explicitly declared being immediate by setting the `immediate` property of `@Component` to `true` or it is declared so implicitly as long as the `class` annotated with `@Component` does not directly implement an `interface`. An immediate DS starts in the state `UNSATISFIED` and is activated as soon as all its mandatory references have been satisfied. Whenever this component gets deactivated, e.g., the declaring bundle is stopped, it is deactivated and goes back into its `UNSATISFIED` state.

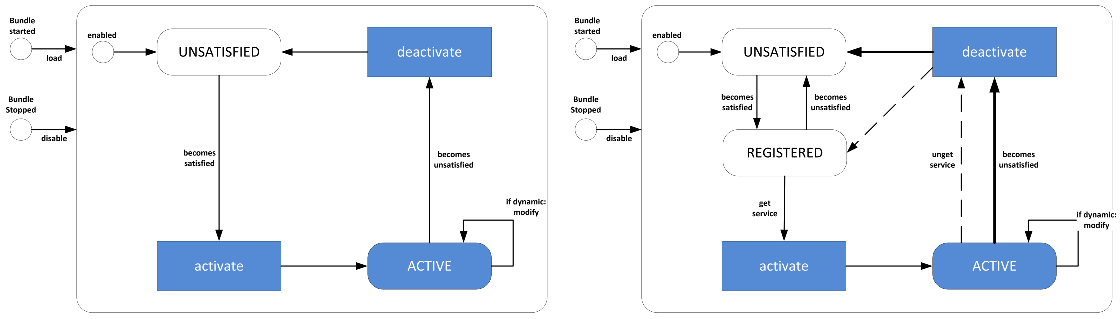

Figure 2.19.: Immediate and delayed component's lifecycle.[37]

This lifecycle is quite different from the delayed DS depicted on the right side. A DS is delayed as soon as the annotated `class` directly implements an `interface` or if the `immediate` property has been explicitly set to `false`. A delayed DS also starts in its `UNSATISFIED` state, but when becoming satisfied is switching to the `REGISTERED` state and is not getting activated yet. Its activation is delayed until another DS requests the delayed DS. In that case the delayed DS also gets activated. Whenever the requesting DS releases the reference to the delayed DS, the delayed DS switches back into the `REGISTERED` state. Apart from that all the rest of the states are transitioned like they are for an immediate DS.

2.4.3.3. Service Component Registry

Until now we have declared several `classes` to be DSs by annotating them with `@Component` but have not looked yet at the facility that takes care of wiring all these components together. This is where the service component registry comes into play which is also the core of every OSGi application.

In part this service component registry can be compared to Dependency Injection (DI) frameworks like Contexts and Dependency Injection (CDI) [38] or Google Guice [39]. DI frameworks are used to lower the complexity of large applications by making use of the principle of "Inversion of Control" [40]. Usually this principle is implemented by outsourcing the wiring of different software parts to an external component and a configuration file instead of letting the components take care of their wiring themselves. The wiring can then be configured to the needs of the application and its changes over time simply are reflected by changing the configuration file. The same principle is also implemented by the OSGi service component registry that takes care of wiring components together according to their declared dependencies via `@Reference` annotated member variables and the components that are available at runtime. Hereby, the service component registry enables DSs to find other DSs without having knowledge of their implementation `classes` but only knowing about the implemented `interface`, i.e., their services. The usage of an `interface` as a mandatory contract between requesting component and requested service highly enhances the reusability of components and meanwhile enforces developers to adhere to best practices in software development.

What makes the OSGi service component registry stand apart from the above mentioned solutions is the support of dynamic dependencies. Dynamic means components that come and go over time, whereas dependencies in Guice for example stay forever. This way a component can react to new components registered at runtime as well as old components being deregistered. A common implementation that makes use of dynamic dependencies is the Whiteboard Pattern[41] that is widely used in OSGi applications.

> *"The whiteboard pattern is defined as a server that uses the OSGi service registry to find its constituents, where each constituent is registered as a service. A constituent can for example be like a traditional listener. This is in contrast with a pattern where the server registers itself as a service and the constituents then register with this server service."*[42]

In Figure 2.20 a graphical description of the Whiteboard Pattern is given with the usual graphical syntax for OSGi services. The rectangular shapes represent bundles, whereas the triangular shape represents a service. Bundles connected to the flat side of the triangle are requesting the service, while bundles connected to the tip of the triangle provide such a service.

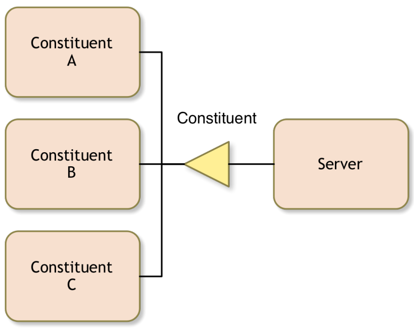

Figure 2.20.: Whiteboard Pattern [42]

One of the most prominent implementations of this pattern is the `HttpWhite` boardService that allows developers to register new Servlets only by adding specific metadata to the DS that defines the path under which the Servlet can be reached. At runtime the HttpWhiteboardService is notified by the service component registry whenever a new component with this specific metadata is registered. The HttpWhiteboardService then takes care of registering this component as a Servlet under the specified path. Whenever this component is deregistered the HttpWhiteboardService is also notified and subsequently deregisters the according Servlet. Thus, it is possible to create a web application that can be changed at runtime simply by (de-)registering components.

2.4.4. Configurable Components

OSGi offers a variety of standardized mechanisms and services that are all defined within their core specification [43] and compendium specification [44]. One of those services is the ConfigurationAdmin which enables developers to reconfigure components at runtime programmatically. Each component has implicitly a configuration that consists of all property values of the annotations used within this component, e.g., one can set a specific reference by altering the target property of a specific @Reference annotation.

What might sound trivial is in fact a mechanism that allows components to be completely runtime reconfigurable, thus making whole systems being able to be configurable at a fine granular level. In Listing 2.12 we depicted a simple example for using the ConfigurationAdmin to reconfigure a component's reference. In this case we force ConfigurableService to use a FooService that has its foo property set to the value foobar instead of bar as the component declares it. This mechanism can be used to adapt existing components to changes that are made after the component has been deployed. If one translates this simle example to the scale of a complete application one is capable of reconfiguring a whole system at runtime.

```
@Component
public class ConfigAdminShowcase {
  @Reference
    private ConfigurationAdmin cm;

    @Activate
    private void activate(){
      Configuration config = cm.getConfiguration("exampleConfig");
        config.update(new HashTable<String, Object>() {{ put("FooService.
            target", "(foo=foobar)"); }};);
    }
}

@Component (pid="exampleConfig")
public class ConfigurableService{
  @Reference(target = "(foo=bar)")
    private FooService fs;
}
```

Listing 2.12: Change reference target via ConfigrationAdmin.

The configuration of a component does not only contain the implicit properties given by the used annotations, e.g., *<ReferenceName>.target*, but also properties that are set by the developer through the `property` property of `@Component`. This way any possible metadata can be altered by other components through the simple usage of `ConfigurationAdmin`, metadata that might influence additional semantics defined by the developer.

Part II.

PRINCIPLES, REALIZATIONS AND EVALUATIONS

3
Common Error Reduction with AADL and RTSJ

3.1. Motivation

It is common knowledge in software engineering that an early development error detection lowers the costs of its correction. This is especially the case if the system under development is an embedded one or a SCSS, whereby not only a system's software, but also its corresponding documentation or hardware is affected by changes.

Although already shown in chapter 1, Figure 3.1 is repeated here to again illustrate the impacts of late error discovery on costs of a software project. Even though 70% of errors are introduced into a system during design phase, where their costs to be removed would be lowest, only 3,5% of them are also discovered during this phase. The vast majority of errors is discovered either during unit, integration and system tests, or even worse during operation and maintenance accompanied with 5 up to 1000 times the costs for removal as compared to design time discovery.

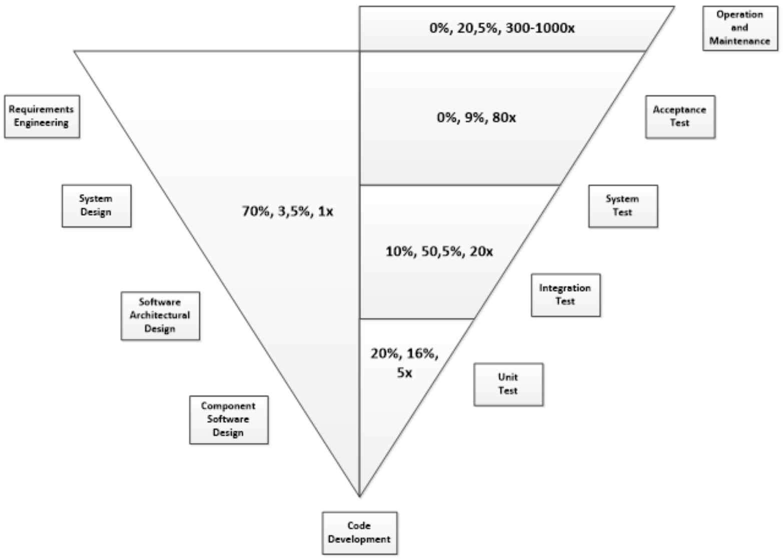

Figure 3.1.: Introduction of errors, their finding and the relative costs compared to errors discovered during design phase. [7, 8]

In order to reduce time and costs related to late error discovery MDD aims at shifting most aspects of a system's software implementation into earlier phases of the development, e.g., software design or system design. This is usually done by establishing thoroughly tested transformations between models of different level of abstraction of the overall system, as well as by introducing code generators that finally convert sufficiently detailed models into executable code.

Although, theoretically the whole system, including business logic, could be represented as models in this work we decided to merely concentrate on shifting structure, timing and inter-component communication aspects of a system's software from the implementation phase to the system design phase. This decision was made because of the increased workload of modeling business logic in comparison to only modeling more abstract aspects of a system, i.e., structure, timing and inter-component communication.

In order to benefit from MDD a sufficiently expressive and preferentially semantically well-defined source language has to be chosen as well as a target language supporting the semantic aspects of this source language. In chapter 1 we already have justified our decision to use AADL as source and RTSJ target language, thus we only explain what additional benefits we expect of those two languages.

The chosen source language, i.e., AADL, offers the following benefits for developers of SCSS:

- Integration of several stakeholders into one central source of information, e.g., requirements, testing, implementation

- Strong type system down to the level of measurement units, e.g., meter vs. yards or gallons vs. liter

- Predefined analyses targeting several aspects of SCSS, e.g., schedulability of the system, fault tree analyses, etc.

- Standardized semantics for all model elements and properties of those

Whereas, our chosen target language, i.e., RTSJ or in a more general sense Java, offers the following benefits over traditional SCSS programming languages, e.g., Assembler or embedded C:

- Reduction of complexity due to language-specific constructs, like `classes`, inheritance and abstraction

- Production-grade real-time garbage collection in contrast to ad-hoc memory management solutions per project or company

- Production-grade standard libraries instead of ad-hoc libraries per project or company

- Highly sophisticated Integrated Development Environment (IDE)s supporting features like autocompletion, instant compile time error recognition, standard refactorings and many more

Our approach uses the standardized semantics of AADL to define a mapping between AADL and RTSJ. By an implementation of this mapping, we generate AADL semantic-compliant RTSJ code which preserves the timing behavior and inter-component communication defined in an AADL model. Thus, a system designer is capable of designing and performing analyses regarding communication and timing almost completely during design phase, while resting assured that the implementation will reflect made design choices. Simultaneously, programmers are relieved of the monotonic and repetitive task of writing communication- and timing-related code.

The rest of this chapter is structured as follows: section 3.2 presents our mapping from AADL to RTSJ which details the mapping concerning structure, timing and communication, as well as the system setup of a system generated with our approach. Finally, we evaluate our approach via a comparison with a handwritten solution.

3.2. Mapping

In this section we present the three main parts of our mapping between AADL and RTSJ. In the first part of this mapping we will show which structural characteristics of AADL are mapped onto the target language. In the second part we present the timing-related features that can be automatically transformed into semantically appropriate RTSJ code. The last part shows how inter-component communication defined in AADL is transformed into inter-thread communication mechanisms in RTSJ.

3.2.1. Structural Mapping

The structural mapping of our approach covers the AADL language constructs type declaration, implementation declaration, **`subcomponents`** and the relationships **`extends`**, **`realizes`** and **`refined to`** that enable a user to structure components and refine them incrementally. In order to keep the following sections readable, we decided not to use a formal representation. Instead, a structural definition is given through the main concepts of AADL.

3.2.1.1. Type and Implementation Declarations

Type declarations in AADL are representing `interfaces` in form of **`features`** that are provided by a realizing implementation declaration. In Java, the concept of an `interface` already exists. An `interface` defines method signatures which have to be realized by `classes` that implement the `interface`. In order to reuse this `interface` concept to reflect the type declaration concept in AADL, we translate each **`feature`** defined by a type declaration into corresponding method signatures. In our approach, we only regard **`data ports`** which can hold one **`data`** element at a time and therefore can be translated into simple `in` and `out` methods. This mapping is shown in Listing 3.1 and Listing 3.2, where the **`in/out data port dataIO`** is translated into its corresponding `in` and `out` methods, each with the declared classifier **`A`** as parameter type.

```
process processA
  features
    dataIO : in out data port A;
end processA
```

```
data A
end A
```

Listing 3.1: Types in AADL

```
public interface ProcessA{
  void inDataIO(A data);
  void outDataIO(A data);
}
public interface A{}
```

Listing 3.2: Types as interfaces in Java

Implementation declarations can **realize** exactly one type declaration and **extend** exactly one other implementation declaration. Multiple inheritance is forbidden by the standard. The extended implementation declaration has to **realize** the same type declaration as the extending one. One type declaration can have several implementation declarations, each inheriting the **features** defined by the type declaration. Additionally, implementation declarations can have **subcomponents** which specify the inner composition of a component. When transferred to Java, implementation declarations resemble `abstract classes` which have member variables that hold references to their **subcomponents**.

```
process implementation processA.impl
  subcomponents
    worker1 : thread threadA.impl;
    worker2 : thread threadA.impl;
end processA

thread threadA
  features
    dataIO : in out data port A;
end threadA

thread implementation threadA.impl
end threadA.impl
```

Listing 3.3: Implementations in AADL

Listing 3.3 depicts a **process implementation processA** of the **process type** defined in Listing 3.1 which inherits the **in/out data port** and adds two **thread subcomponents**, **worker1** and **worker2**. This implementation declaration is translated into a Java `class` as shown in Listing 3.4. The `class ProcessAImpl` implements the `interface ProcessA`, inherits the methods defined by `ProcessA` and implements them. The implementation of `in/out` methods is explained in detail in subsubsection 3.2.3.3. Likewise, each **subcomponent** is represented by its own member variable, i.e., `worker1`, `worker2` and uses the declared **classifier** as type, i.e. ThreadAImpl. The member variables are initialized within the constructor as all **classes** and **interfaces** can be seen as blueprints which have to be assembled by an outside entity.

```java
public abstract class ProcessAImpl implements ProcessA{
  ThreadAImpl worker1;
  ThreadAImpl worker2;

  public ProcessAImpl(ThreadAImpl worker1,
        ThreadAImpl worker2){
    this.worker1 = worker1;
    this.worker2 = worker2;
  }

  @Override
  public void inDataIO(A data){
    ...
  }

  @Override
  public void outDataIO(A data){
    ...
  }
}

public interface ThreadA{
  void inDataIO(A data);
  void outDataIO(A data);
}

public class ThreadAImpl implements ThreadA{...}
```

Listing 3.4: Implementations as classes in Java

3.2.1.2. Hierarchies and Refinements

Type declarations and implementation declarations can be used to create an inheritance hierarchy of components by letting one component extending another or by an implementation declaration realizing a type declaration. The **extends** relation of AADL only allows single inheritance and demands the extending component to be of the same type, e.g., **system**, **process**, etc., as the one that is extended. Therefore, we can simply reuse the extends keyword from Java in order to map this relation. Type declarations inherit all **features** from their parent type declaration which is mapped in Java by the child interface inheriting all method signatures from the parent interface. The same concept can be transferred to implementation declarations that are classes instead of interfaces in Java. An implementation declaration extending another implementation declaration inherits all **features** and **subcomponents** from its parent. In Java, a subclass extending a superclass inherits all non-private member variables, representing **subcomponents** and all methods representing **features**.

In case of an implementation declaration realizing a type declaration, we will use the implements keyword in Java to map the semantic meaning. A realizing implementation declaration inherits all **features** defined by the type declaration. The same semantic meaning is given by a Java class that is implementing an interface, whereby all methods declared in the interface are inherited and implemented by the class.

Problems arise when it comes down to the **refine** mechanism in AADL. The refinement of a component encompasses, among other things, the possibility to refine classifiers of **ports** by an extending type declaration. In order to reflect the classifier refinement of a **data port** in an extending and refining type declaration, we have to change the signature of its corresponding method in Java. Listing 3.5 shows a simple refinement of an **in data port**, where the classifier gets specialized by the extending type declaration. In Java, this would lead to an interface definition as shown in Listing 3.6. Declared as depicted, the overridden in method for dataIO is an invalid method signature. In Java, an overriding method must have the same signature composed by method name and parameter types as declared in the super type. As the overriding method changes the parameter types, it is not valid.

In order to restrict the possible types for refinements, AADL defines the property **Classifier_Substitution_Rule** which can be associated with a **port**. Currently, AADL defines three different values for this property, **Classifier_Match**, **Type_Extension** and **Signature_Match**. We only consider the former two. **Classifier_Match** – the default value – enforces the refined classifier to be exactly the same as the classifier in the extended component. **Type_Extension** allows the refined classifier to be a subtype of the one used by the extended component.

```
thread threadA
  features
    dataIO : in data port A
    {Classifier_Substitution_Rule=>Type_Extension};
end threadA;

thread threadB extends threadA
  features
    dataIO : refined to in data port B;
end threadB;

data A
```

```
end A

data B extends A
end B
```

Listing 3.5: Refinement of features in AADL

```
public interface ThreadA{
  void inDataIO(A data);
}

public interface ThreadB extends ThreadA{
  @Override
  void setDataIO(B data); //Compiler Error
}

public interface A{}
public interface B extends A{}
```

Listing 3.6: Erroneous refinement of features in Java

Addressing the previously mentioned problem of invalid method overriding, we decided to use Java's `default` implementation mechanism [45] to prevent an illegal use of the wrong method for a refined classifier of a **`data port`**. As depicted in Listing 3.7, the refined `in` method is defined in `interface ThreadB` and the now illegal `in` method, that is inherited from `ThreadA` is per default redirecting to the newly defined method. If the given parameter does not match type `B` a `ClassCastException` will be thrown. This way we conform to the semantic meaning of the AADL model, while only making minimal changes to the inheritance mechanisms in Java.

```
public interface ThreadA{
  void inDataIO(A data);
}

public interface ThreadB extends ThreadA{
  @Override
  default void inDataIO(A data) {
        inDataIO((B)data);
  }

  void inDataIO(B data);
}
```

Listing 3.7: Valid refinement of features in Java

3.2.2. Timing concerned Mapping

In this section, we explain how the different semantics for timing of **`threads`**, **`data ports`** and their **`connections`** are mapped from AADL to RTSJ.

As already described in subsubsection 2.2.2.3, **`threads`** are the core concept for running software in AADL. As such they have a semantic defined by AADL that describes their timing behavior in detail. We restrict the **`Dispatch_Protocol`** of **`threads`** in this work to **`periodic`**, so we will only explain the behavior of **`periodic`** **`threads`** and properties altering it.

3.2.2.1. Period, Deadline and Priority

`Periodic` **`threads`** must have a period, i.e., an interval at which they are executing code. The period can be given by the property **`Period`** in form of a number and a time unit, e.g., 200 ms or 3 sec. **`Period`** can be directly translated into RTSJ by using a `PeriodicTimer` for which a period can be given. A `Timer` is an event trigger and meant to be used in conjunction with the aforementioned `AsyncEventHandlers`, so the code to be executed is encapsulated in the `handleAsyncEvent()` method of the handler.

Another timing-related AADL property is the **`Compute_Deadline`** of a **`thread`** which states until when the computation has to be done at the latest. In RTSJ, an `AsyncEventHandler` can have so called `ReleaseParameters` which define a deadline and a `deadlineMissHandler` that is called in the case of a not fulfilled deadline.

Although not actually being a timing-related property, the **`Priority`** is nevertheless essential for every priority-based scheduler. It can be given for each **`thread`** and can be represented by `PriorityParameters` in RTSJ which again are associated with `AsyncEventHandlers`. Listing 3.8 and Listing 3.9 depict the mapping between a **`thread implementation`** defined in AADL, with the three mentioned properties associated, and a `class` in RTSJ representing an instance of this **`thread implementation`**.

```
thread implementation threadA.impl
  properties
    Dispatch_Protocol => periodic;
    Period => 200ms;
    Priority => 5;
    Compute_Deadline => 100ms;
end thread.impl;
```

Listing 3.8: Timing in AADL

```
public class ThreadAInstance extends ThreadImplA{
  private AsyncEventHandler handler = new InnerAsyncEventHandler();
  private Timer timer = new PeriodicTimer(null,new RelativeTime(200,0),handler
      );

  class InnerAsyncEventHandler extends AsyncEventHandler{

    public InnerAsyncEventHandler(){
      setDeamon(false);
      setSchedulingParameters(new PriorityParameters(5));
      ReleaseParameters rps = timer.createReleaseParameters();
      rps.setDeadline(new RelativeTime(100,0));
      setReleaseParameters(rps);
    }

        @Override
    public void handleAsyncEvent(){ //Logic }
  }
}
```

Listing 3.9: Timing in RTSJ

By default, all `AsyncEventHandlers` are treated as daemons, i.e., background tasks that are only executed as long as the main thread in Java is running. In order to make an `AsyncEventHandler` a foreground task which is executed independently from the main thread we have to explicitly call `setDeamon(false)`. The `SchedulingParameters` of `ThreadAInstance's` `InnerAsyncEventHandler` are used to reflect the **`Priority`** of **`threadA.impl`**. The **`Dispatch_Protocol`** and **`Period`** are directly translated into a `PeriodicTimer` with its period set to 200 ms. The handler for this timer is `InnerAsyncEventHandler` as it extends `AsyncEventHandler`. Finally, the `ReleaseParameters` are used to map the **`Compute_Deadline`**.

3.2.2.2. Input_Time and Output_Time at Data Ports

Timing is not only important in the context of **`threads`** and their execution time, but also for **`ports`** and the time when they are receiving or sending data. Based on **`Input_Time`** and **`Output_Time`** property values given for a **`port`**, there are basically two possibilities for **`data ports`** in AADL.

First, a **`data port`** receives/sends **`data`** at a given lifecycle event of the **`thread`** it belongs to, i.e., **dispatch**, **start**, **completion** or **deadline**, without any additional offset. In this case, the receiving/sending can be done by the **thread** itself.

Second, the **data port** defines its timing with a given offset in relation to one of the above mentioned lifecycle events. In this work we only consider a positive offset as sensible. In this second case, the **thread** is no longer able to do the receiving/sending on its own, but has to start a parallel task at the given lifecycle event. This parallel task then executes at the given offset and receives/sends data from/to a **data port**.

```java
public class ThreadAInstance extends ThreadAImpl{
  private InDataPort<Object> dataIn;
  private InDataPort<Object> dataInWithOffset;
  ...

  private final void dispatch() {
    datain.receiveInput();
    new Handler(dataInWithOffset);
  }
  private final void start() {...}
  private final void compute() {...}
  private final void completion() {...}

  class InnerAsyncEventHandler extends AsyncEventHandler{
    ...
    public void handleAsyncEvent(){
      dispatch();
      start();
      compute();
      completion();
    }
  }
}

public class Handler extends BoundAsyncEventHandler{
  private InDataPort<Object> dataIn;

  public Handler(InDataPort<Object> dataIn){
    this.dataIn = dataIn;
    setSchedulingParameters(new PriorityParameters(5));
    Timer timer = new OneShotTimer(new RelativeTime(30,0),this);
    timer.start();
  }

  !@Override
  public void handleAsyncEvent() { dataIn.receiveInput(); }
}
```

Listing 3.10: Data port timing in RTSJ

In Listing 3.11 two exemplary **data ports** are given. First, an **in data port** with no explicitly defined **Input_Time** which is set to **Dispatch** and 0 ns offset per default. Second, an **`in data port`** with an **`Input_Time`** set to **`Dispatch`** and a positive offset between 30 ms and 40 ms. In Listing 3.10 the timings of those two **`in data ports`** are translated into the direct method call `datain.receiveInput()` and a parallel handler. The handler executes the same method with an offset of 30 ms which is the lower bound of the offset specified in the AADL model. The **`Priority`** of the handler is set to the same value as the **`thread`** that creates it, i.e., 5.

```
thread threadA
  features
    dataIn : in data port;
    dataInWithOffset : in data port
    {Input_Time => ([Time => Dispatch;
    Offset => 30ms .. 40ms]);};
  properties
    Priority => 5;
end threadA;

thread implementation threadA.impl
end threadA.impl
```

Listing 3.11: Data port timing in AADL

3.2.2.3. Immediate, Delayed and Sampled Connections

Although a specific **`Input_Time`** or **`Output_Time`** can be given for a **`port`**, the **`connection`** between two **`ports`** also has implicit effects on the timing aspects of a **`port`**. This implicit behavior is expressed by setting the **`Timing`** property of a **`connection`** between two **`data ports`** of two **`periodic threads`** to **`immediate`**, **`delayed`** or **`sampled`**.

By default, all **`connections`** are set to be **`sampled`** as this has no effects on the timings given directly via **`Input_Time`** or **`Output_Time`**. The receiving **`thread`** would always receive the latest data from the sending **`thread`**.

For **`immediate connections`**, the **`Input_Time`** of the receiver is forced to be start as **`IO_Reference_Time`** and zero offset. The **`Output_Time`** for the sender is assumed to be completion as **`IO_Reference_Time`** and also zero offset, but can be overridden if a single value for **`Output_Time`** is given. An **`immediate connection`** enforces the receiver to be delayed until the sender completes execution. This ensures predictable communication within one dispatch frame as depicted in Figure 3.2. In RTSJ this behavior is enforced by using a common synchronization object for the **`immediate connection`** on which the receiving thread calls `wait()` as long as the call to `isDirty()` of the corresponding port returns `false`. The sending thread then wakes up the receiving thread after writing a new value to the respective port, by calling `notifyAll()` on the common synchronization object.

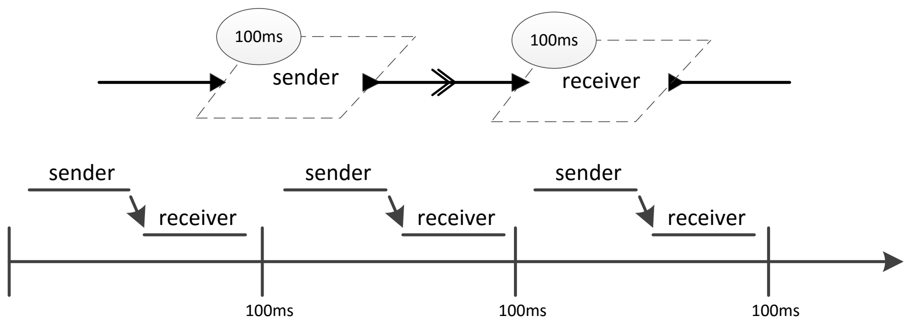

Figure 3.2.: Communication via Immediate Connection [19]

`Delayed connections` initiate the transmission of data at the deadline of the sender, thus having an **`Output_Time`** with deadline as **`IO_Reference_Time`** and zero offset. Accordingly, the receiver has an **`Input_Time`** with dispatch as **`IO_-Reference_Time`** and zero offset as well. This way, the data is received at the next dispatch of the receiver following or equal to the sender's deadline as depicted in Figure 3.3.

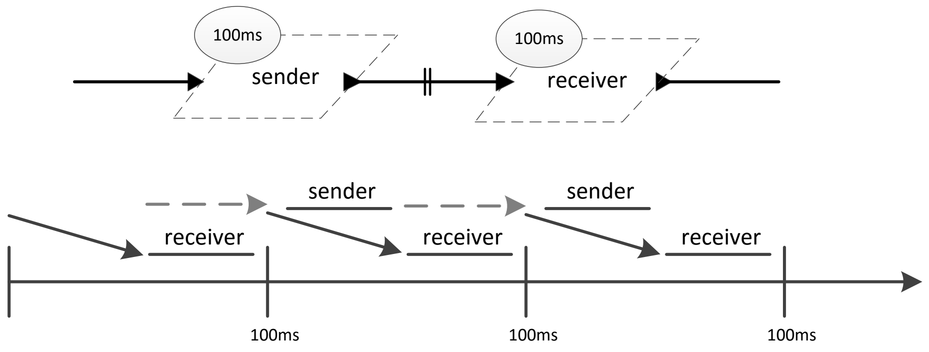

Figure 3.3.: Communication via Delayed Connection [19]

3.2.3. Communication-related Mapping

AADL defines several mechanisms concerning communication between components. Regarding our employed subset of AADL, we only consider **`port`** connections and the property Classifier_Matching_Rule. First, we will explain in detail how the mentioned property is semantically defined and subsequently how the mapping of modeled port connections into RTSJ code is done.

3.2.3.1. Classifier_Matching_Rule

The Classifier_Matching_Rule property defines how the classifiers of two connected data ports must conform to each other. Two possible values are currently of interest for our approach, i.e., Classifier_Match and Type_Extension. Other possible values that are not covered by our work are Equivalence, Subset and Conversion. Classifier_Match is the simplest case whereby the classifier of the source port has to match exactly the classifier given at the destination port. Classifier_Match is also the default value applying to every connection if not specified otherwise. The rule Type_Extension enforces the destination port to have a classifier that is either the same as the one of the source port or a subtype, e.g., a data type declared to be extending the source's classifier. By default, both possibilities are covered by the extends semantics of Java. The mapping declared in subsection 3.2.1 automatically leads to valid Java code, regarding the classes used as parameter types in methods that represent ports.

3.2.3.2. Port Connections

Port connections in AADL are always defined within a component implementation declaration. Each connection has a source port and a destination port. In Figure 3.4 a process with two thread subcomponents is depicted where all components are connected via directed port connections. For our mapping we identified two categories of port connections within a given component implementation declaration. First, port connections that have a subcomponent's port as destination, e.g., downward or subcom_con1. Second, port connections that have a port of the component itself as destination, e.g., upward.

In order to map these two categories of connections, we decided to generate a separate class – a Connection Broker (CB) – for each component that takes care of transmitting data over connections declared within that component. Listing 3.12 exemplarily shows a CB for the process depicted in Figure 3.4. The

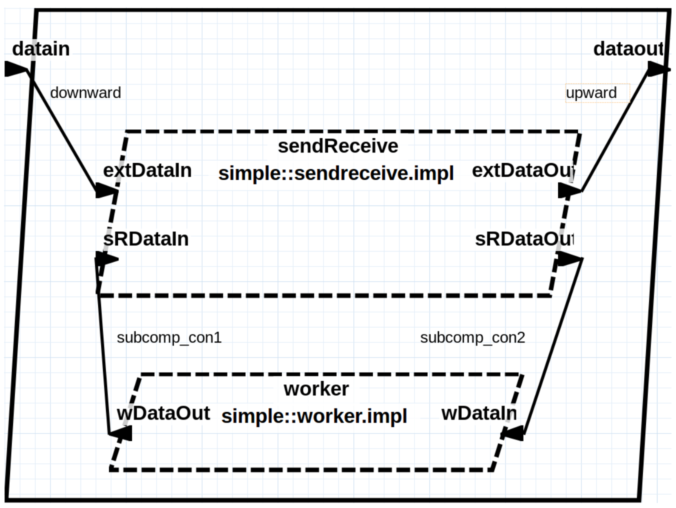

Figure 3.4.: Port connections within a component

CB has a member variable `myself` for the **`process`** it manages the **`connections`** for as well as for each **`subcomponent`** of `myself`, i.e., `sendReceive` and `worker`.

The actual transmission of data takes place in the method `sendOnConnection()`, where the name of the **`connection`** is passed as a unique identifier. If called, the method decides – based on the **`connection's`** name – which corresponding in/out methods of the component or one of its **`subcomponents`** have to be called. Depending on the chosen in/out method's parameter types, the transmitted data may have to be cast to the corresponding type.

In Listing 3.12, the case `"subcomp_con1"` is a representative of the first of the two above-named **`connection`** categories and represents the eponymous **`connection`** in Figure 3.4. As the destination **`port`** of this **`connection`** is `sRDataIn` of the **`thread subcomponent`** `sendReceive`, the corresponding method `inSRDataIn()` is called on the **`subcomponent`** member variable `sendReceive`. The port declares the base type `Boolean` as classifier, therefore the transmitted data is cast to this type.

The second category of **`connections`** is represented by the case `"upward"` in Listing 3.12. Also representing the eponymous **`connection`** in Figure 3.4, its destination is the **`out data port`** `dataout` of the component itself. Thus, the respective `out` method of `myself` is called. The given data is cast to `Boolean`, as declared by the **`out data port`** `dataout`.

```java
public class ConnectionBroker{
  private SomeProcessImpl myself;
  private SendReceiveImpl sendReceive;
  ...

  public ConnectionBroker(Componet myself, SendReceiveImpl sendReceive,...){
    this.myself = myself;
    ...
  }

  public void sendOnConnection(String connection, Object data){
    switch (connection) {
      case "subcomp_con1":
        sendReceive.inSRDataIn((Boolean)data);
        break;
      ...
      case "upward":
        myself.outDataout((Boolean)data);
        break;
    }
  }
}
```

Listing 3.12: Connections in RTSJ

3.2.3.3. In and Out methods

As explained in subsubsection 3.2.3.2, a CB is a sufficient possibility to manage the transmission of data within a component. In case of communication inside a **subcomponent** or outside of the given component, a CB calls the in/out methods of the respective component. The in/out methods take care of further routing the given data to their destination. The in methods of **subcomponents** handle this routing via their own CB, but in order to forward the given data over an **out port** of the component itself, a component has to use the CB of its parent component. Therefore, each component has – in addition to its own CB `broker` – a member variable `parentBroker`, as depicted in Listing 3.13. Below, the implementation of `in` and `out` methods of a generic component are explained in detail.

For `in` methods of a component, there are two possibilities. The first is, there are outgoing **connections** for the given **in data port** within the component. Thus, the method forwards the incoming data via the component's CB as shown in the method `inMethodForwarding()` in Listing 3.13. The second is, there are no outgoing **connections** declared for the given **in data port** within the component. Then, the respective `in` method stores the incoming data within a designated **port** member variable, e.g. `inPort`, as depicted by the method `inMethodFinal()`.

For `out` methods, there is only the possibility of forwarding the given data via the `parentBroker`. An out method resembles a broadcast as it sends the given data on all outgoing **`connections`** that are declared within its component's parent component. This is done via the `sendOnPort()` method of the `parentBroker`. This method works similar to the `sendOnConnection()` method, as shown in Listing 3.12. Merely the unique identifier is different, as several **`subcomponents`** of the parent component might have the same name for their **`out data ports`**. Thus, the concatenation of the components name and its **`out data port's`** name is used.

```java
public class Component{
  private ConnectionBroker broker;
  private ConnectionBroker parentBroker;
  private InDataPort<Object> inPort = new InDataPort<Object>();

  public void inMethodForwarding(Object data){
    broker.sendOnConnection("con1", data);
    ...
    broker.sendOnConnection("conX", data);
  }

  public void inMethodFinal(Object data){
    inPort.setFWData(data);
  }

  public void outMethodForwarding(Object data){
    parentBroker.sendOnPort("compName+portName", data)
  }
}
```

Listing 3.13: In and out methods in RTSJ

3.2.4. System Setup

In order to provide a developer with an easy way to start (and test) the generated system, our approach also encompasses the generation of a central `Main class`, which implements the mandatory `main` method that is the entry point for every Java application. Within this `main` method all generated `classes` are instantiated and assembled according to their hierarchical structure defined in the AADL model, e.g. if a developer defined a **`process A`** with two **`thread subcomponents T1`** and **`T2`**, then within the method there will be three statements as depicted in Listing 3.14.

```java
public class Main{
  public static void main(String[] args){
    ThreadT1 t1 = new ThreadT1();
    ThreadT2 t2 = new ThreadT2();
    ProcessA a = new ProcessA(t1, t2);
  }
}
```

Listing 3.14: Simplified main method for a system defined in AADL

The constructor of `ProcessA` is generated in such a way that it accepts two parameters of the type `ThreadT1` and `ThreadT2`, which represent the **`subcomponents`** of `ProcessA`. As `ThreadT1` and `ThreadT2` do not define any **`subcomponents`** of their own, their constructors are parameterless. However, this depiction only represents a simplified version of an actual `Main` `class`. In order to avoid naming conflicts regarding `class` names we have to use their fully qualified `class` names, e.g., for `ThreadT1` we have to use its fully qualified `class` name `de.unia.smds.examplesystem.processa.threadt1.ThreadT1`. Additionally, the used variable names for each instance have to be unique, e.g., `t1-12345` where `12345` is a unique identifier created by Java. Unfortunately, this results in not easily legible code as exemplarily shown in Listing 3.15.

```java
public class Main{
  public static void main(String[] args){
    de.unia.smds.examplesystem.processa.threadt1.ThreadT1 t1-12345 = new de.
        unia.smds.examplesystem.processa.threadt1.ThreadT1();
    de.unia.smds.examplesystem.processa.threadt2.ThreadT2 t2-12345 = new de.
        unia.smds.examplesystem.processa.threadt2.ThreadT2();
    de.unia.smds.examplesystem.processa.ProcessA a-12345 = new de.unia.smds.
        examplesystem.processa.ProcessA(t1-12345, t2-12345);
    ...
  }
}
```

Listing 3.15: Actual main method code

Finally, the `main` method contains code needed to establish communication between all components, as well as code to start all defined `threads`.

3.3. Related Work

In the context of AADL, a lot of work was done regarding code generation for different target platforms like [46] or [47]. [46] focuses on AADL and Simulink [48] for modeling architecture and behavior and then generating the corresponding Ada and SPARC code. [47] is a standalone AADL model processor that supports code generation, targeting C real-time operating systems and Ada for native and Ravenscar targets. However, as we decided to use RTSJ as target language, we concentrate on work that has the same target language or at least Java without the real-time capabilities of RTSJ. To the best of our knowledge, we are aware of three different approaches, namely [49], [50] and [51].

[50] focuses on medical applications and a generation of AADL system models into Java code for a given reference platform. The generated code has to be compliant with the specified publish/subscribe mechanisms of the underlying middleware. As reference platform, they use an open source "Medical Application Platform" called "Medical Device Coordination Framework". Although some of the presented generation mechanism are relevant for our work, the highly specialized target platform contradicts our more general-purpose approach, as we use plain RTSJ without any further assumptions about the underlying platform.

[49] indeed uses RTSJ as target language, but focuses on the partitioning of systems defined in AADL for ARINC 653 compliant systems and how this partitioning can be maintained in the generated code. Moreover, a major part of the work deals with communication between partitions and is not concerned with a generalized generation approach.

[51] in contrast is very similar to our approach regarding their goal of a general mapping between AADL and RTSJ. However, their work is simplifying most of the aspects that our work investigates in detail. To allege an example, they simplify the `data port` communication to always happen either at dispatch, start or deadline. The properties *Input_Time* and *Output_Time* are completely ignored as well as timings dictated by the `Timing` property of `data port connection`. Threads are not forced to run consecutively if a `data port connection` between them is marked as `immediate`. Another distinction to our work is the targeted version of RTSJ. While they are targeting version 1.0.2, we use mechanisms from version 2.0 which facilitates the realization of semantics determined by AADL.

3.4. Evaluation

The evaluation of our approach follows the layout explained in section 2.1, i.e., is split into a quantitative and a qualitative part. As we are only interested in the parts of code that are related to either structure, timing or communication we stripped the examples of all logic-related parts, i.e., no autopilot logic is contained. In order to evaluate the generated code we assume a handwritten, golden-standard solution that is designed simple in terms of structure and efficient in terms of communication patterns, e.g., all communication is assumed to be executed via a direct method call which is the best possible communication time-wise.

First, a quantitative comparison between the handwritten code and the generated code was done. Hereby, we examined the differences between both solutions regarding metrics like Lines of Code (LoC), performance in terms of latency introduced by the communication-related code and, most important, reduction of complexity. Second, a qualitative evaluation was done by executing different scenarios that are common in a day-to-day development workflow.

3.4.1. Quantitative Evaluation

For the quantitative part of our evaluation we examine four different metrics and how those differ between the handwritten code base and a code base compliant to our mapping.

In embedded systems memory often is a restricting factor, therefore the knowledge about size of written code is crucial for a successful deployment of software to a given target platform. Therefore, the first metric are SLOC (without comments) as those usually are a good indicator for the final size of the machine code that is deployed to the target system. In Figure 3.5 we depicted the SLOC of both solutions first the manual written ones and then the generated ones. Regarding this metric our approach adds roughly 4-5 times the amount of code in comparison to a handwritten solution. The majority of this massive overhead stems from the more complex communication patterns used by the generated solution. Where the handwritten solution relies upon direct calls between different system components, the generated one offers a more generic but also more verbose solution that enables it to deal with different types of communication, e.g. immediate or delayed ones.

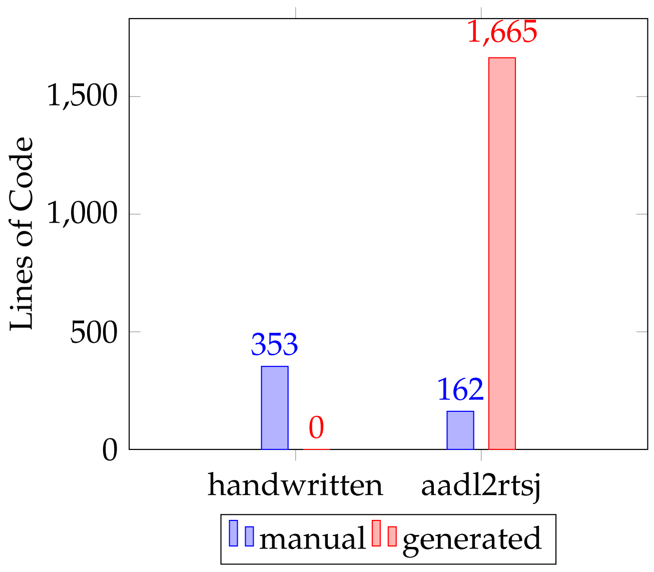

Figure 3.5.: SLOC comparison between handwritten and aadl2rtsj solution

However, it should not be forgotten that the manually written model code in the generated approach is roughly half the size of the handwritten solution. The SLOC observed in Figure 3.5 led us to the conclusion that the initial time of writing would be roughly half the time required for a handwritten solution. This speedup might even be further amplified by how Java code usually is structured in contrast to how AADL code. In Java each component as well as its type are separated into different `classes` and `interfaces`, thus into different files. In AADL instead, the whole code resides within one file. The creation of each of the mentioned Java files, just as the navigation between those, generates a non-neglectable amount of work that has to be done by a programmer, which in turn leads to an additional increase in time it takes to write the code.

After examining the observable effect of our approach on SLOC and thus time of writing we now turn to the performance of our generated code. As structure and timing code are virtually the same in handwritten and generated solution we focus on the communication-related code of both solutions.

In order to compare both solutions we first created two benchmarks. The benchmark for the handwritten solution assumes the developer to know exactly which component has to be called and thus results in a direct method call. The benchmark for the generated solution is a little more complex as the generated solution makes heavy use of large switch-case statements in its code to decide

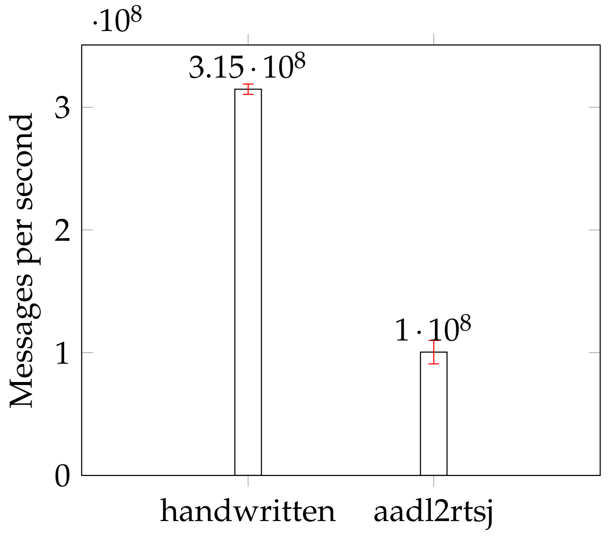

Figure 3.6.: Comparison of message throughput between handwritten and aadl2rtsj solution

which component shall receive the message or if the message shall be delegated to another CB as depicted in Listing 3.12.

In Figure 3.6 we depicted a simplified comparison between the throughput of our handwritten and our generated solution. According to this figure the generated solution roughly copes with one third the throughput of the handwritten solution. However, this is a simplification of the behavior of a CB in the generated solution. The detailed behavior of a CB, i.e., a switch-case statement in general, can be seen in Figure 3.7. Obviously it depends on the position of the case statement within a switch clause as well as on the call depth, how long it takes the CB to transmit a message successfully. The aforementioned $1 * 10^8$ messages per second correspond to a case position of 5 and a call depth of 5, whereas a call depth of 1 and case position of 5 would already provide a doubling in throughput. Therefore, it depends on how deep an AADL model is nested and how many different `connections` have to be served by one port to determine how fast exactly the transmission of a message in the generated solution is. We assume $1 * 10^8$ messages per second to be a reasonable average. Depending on communication's share of the whole system this decrease in message throughput might have a measurable impact on the overall system performance.

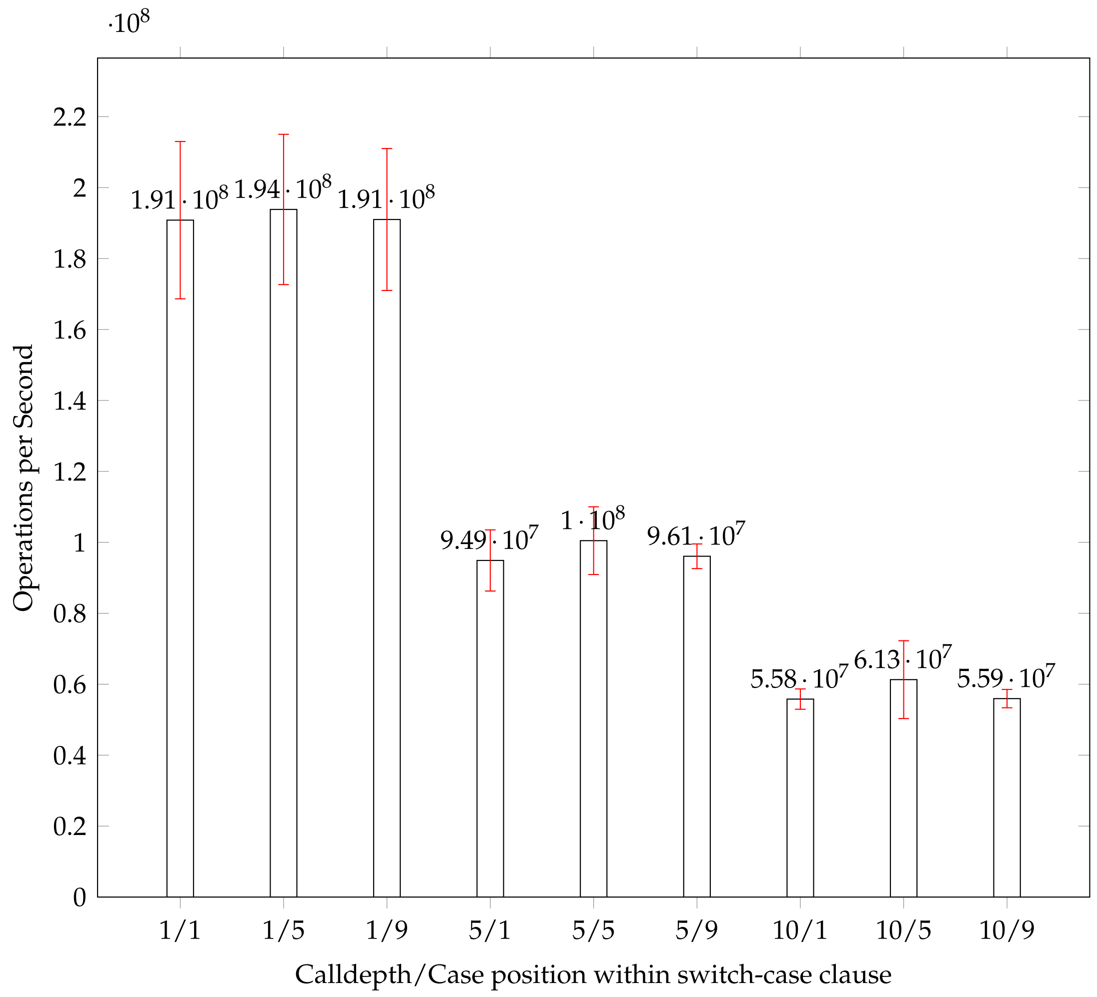

Figure 3.7.: Message throughput with different calldepth/case position combinations

3.4.2. Qualitative Evaluation

In subsection 3.4.1 we compared our solution to a handwritten one based on performance-related numbers. One drawback noted was the increased size of generated code.

Reduced Complexity This drawback of a generated solution is more than compensated for by the reduction of overall complexity as it is depicted in Fig-

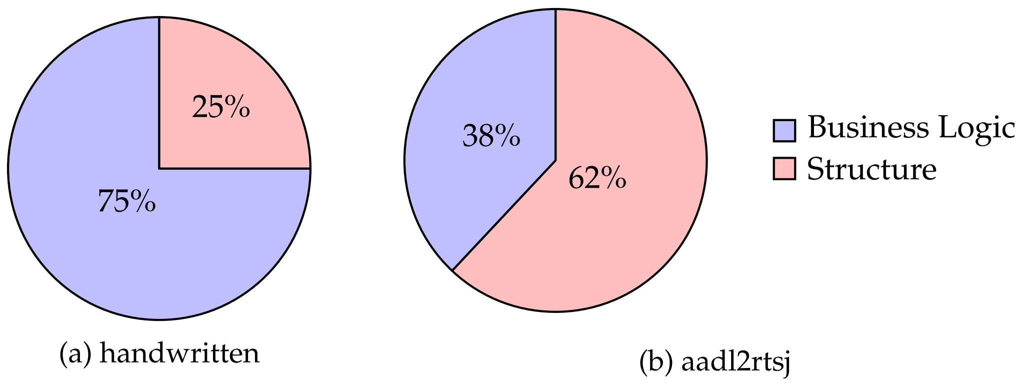

Figure 3.8.: Logic SLOC vs. Structure SLOC comparison between handwritten and aadl2rtsj solution

ure 3.8. Here we define the reduction of complexity as the share of logic-related code of the overall system. This share we measure by counting the SLOC of business logic and structure/timing/communication. The overall system is the sum of both. This way, depending on whether the handwritten or generated solution is taken as reference, the reduction of complexity varies between 25% and 62%. Likely, a real-life system will be somewhere in between as neither it will be simplistic in terms of communication, structure and timing as the current handwritten solution is, nor it will contain as little logic-related code as our generated solution does. Nevertheless, we estimate a complexity reduction of approximately 33% merely by the fact that a programmer does not have to take care of communication, structure or timing of components defined in a system.

Enhanced Reliability The other major advantage of our solution is the quality and therefore safety of the generated code. In a real-world use-case the generator would be one of the most frequently used tools of a developer of a SCSS and therefore be a thoroughly tested solution. Therefore, all structure/timing/communication-related code fragments can be trusted to be working correctly if designed correctly in AADL und subsequentially generated by our production-tested generator. Thus, errors resulting from this code would result from a flawed system design in AADL which already can be detected during design time through extensive model analysis provided by the standardized AADL tooling. This is also shown by the use-case described in the following.

From the running example presented in section 2.1 we pick two components that are communicating via transitive `connections` with each other, i.e., `mix-`

`ThrottlesControl` and **`simulation`** which are transitively connected via the different controllers for position, pitch, roll, etc.. Assume the case a developer wants to add logging logic to the overall system which shall log all incoming commands sent from the **`positionCmd`** and **`positionInfo ports`** of **`simulation`** as well as all resulting throttle values coming from the **`throttle port`** of **`mixThrottlesControl`**.

In a pure simplified Java solution the developer would have to execute the following steps:

- Create a new logger `class`, e.g., `Logger`, implementing a corresponding `interface` to enforce proper encapsulation.
- Write timing logic, regarding initialization, periodic dispatching, stopping, etc.
- Implement a method `void logging(PosCmd cmd, PosInfo info)` which contains the business logic of our logger.
- Alter the implementation `class` of **`simulation`** to keep a reference to the newly created `Logger`.
- Alter the implementation `class` of **`mixThrottlesControl`** to keep a reference to the newly created `Logger`.
- Alter the implementation `class` of **`simulation`** to also call the `logging` method of our newly created logger.
- Alter the implementation `class` of **`mixThrottlesControl`** to also call the `logging` method of our newly created logger.
- Alter the system startup code to create a new instance of `Logger` as well as alter the instantiation of **`simulation`** and **`mixThrottlesControl`** to encompass the passing of a proper `Logger` reference to them for later use.

All these actions have to be done by a developer by hand and no IDE support is given to him when it comes to thinking of all the places the existing code needs an alteration due to newly added entities. Also, no support is given by an IDE regarding (existing) structure, timing or communication in general. Although the code produced is rather repetitive for each new component and its communciation, those actions are still very time consuming as they have to be done by hand for each component individually.

In contrast all a developer has to do – following the modeling approach presented in this chapter – is as follows:

- Create a new `thread` component, maybe inheriting from an already defined abstract `thread`.
- Add three `ports` for the incoming data.
- Add `connections` from the `ports` of `simulation` and `mixThrottlesControl` to the newly created logging component.
- Generate structure/timing/communication related code and write business logic for the newly created logging component.

All those tasks can be done in a graphical AADL IDE that supports the developer with optical feedback, e.g., when a `connection` is missing or a `port`. The only thing left to the developer to be done by hand is to write business logic within his newly created logging `thread`. This business logic can be preserved even if the communication patterns change as long as the data transmitted stays the same. All structural, communication- and timing-related code is generated and also can be regenerated if changes occur. As the generated code is assumed to be safer and less error-prone than code written by hand, this adds an tremendous amount of reliability to code produced with our approach in contrast to a handwritten solution that also adds way more complicated steps to the addition of a simple component than our approach does.

3.5. Conclusion

In this part we presented a mapping approach from an AADL subset to RTSJ which maintains the semantics given by the AADL standard. This approach enables developers of SCSS to shift structure, timing and communication-related concerns into design phase. Hence, they are able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices. The application of our approach is shown via the implementation of an autopilot for quadrocopters. For this purpose the software of the quadrocopter is modeled in AADL and is then generated by our implementation. The use-case shows three advantages of our approach over an implementation without code-generation:

- The speed-up of development by letting the programmer focus on application logic instead of writing recurring code concerned with timing and communication.
- A less error-prone transition from the design of a system to its implementation.
- The possibility of an earlier detection of timing- or communication-related errors in the system.

Additionally, the aforementioned benefits also come together with an enhanced changeability of the overall system. Every time a system changes all a developer has to do is to regenerate it and add missing business logic, instead of rewriting large parts of an existing system by hand and thereby thinking of all possible impacts and variability points that have to be changed too.

Taken some steps further, this approach can lead to the possibility of simple Java developers writing real-time systems designed by one competent real-time system designer. In the next part we will investigate a broader subset of AADL as well as we tackle the yet untouched issue of maintainability.

4
Maintainability with AADL and OSGi

4.1. Motivation

In chapter 3 we presented a mapping from AADL to RTSJ which tackled the challenge "Early Errors and Late Discovery" defined in subsection 1.2.1. By enabling developers of SCSS to shift several aspects of coding into earlier phases and offering them means to perform analyses during those, the aforementioned problem was addressed in major parts. However, one shortcoming of this approach is the monolithic nature of the generated system which is due to the use of plain RTSJ with no modularity concept in mind. Lastly, the other challenges mentioned in section 1.2 remain completely untouched by the current approach.

Therefore, in this chapter we will drop the plain RTSJ-based approach in favor of an RTSJ and OSGi-based one. This step is taken on the basis of our findings presented in chapter 1, where we interrelated the development of CGSS with SCSS. By using OSGi we try to reproduce the observed trend of CGSS, i.e., to move from monolithic applications towards fine granular microservices, on a small scale - within a single JVM.

The aim of this switch from RTSJ to OSGi is mainly to tackle the second challenge defined in section 1.2: "Maintainability". However, by increasing the modularity of a system not only maintainability is improved, but also other aspects that have a positive effect on the remaining challenges "Recertification" and "Variability" as described in the following.

First, maintainability is one of the most important requirements a SCSS has to fulfill. Software that is written for airplanes often has to run not only for a couple of years but decades. The same holds true for banking systems or embedded software, e.g., in satellites. A system that is planned to be run for such a long time must be designed maintainable so that it can stay affordable. Modularity of a system is one major factor that attributes to maintainability. If a system is decomposed into several modules, each with its well-defined `interfaces` and dependencies, then maintenance usually is restricted to only a few modules as the `interfaces` between modules pose a natural endpoint for chain reactions. If in contrast a system is rather monolithic and as such contains a lot of inner dependencies, then a change or a later addition might result in unpredictable side-effects somewhere else in the code.

Second, variability is the next challenge that is affected by the modularity of a system. Given that a system is composed of fine granular modules, each with its well-defined `interface`, then the implementations that are hidden behind those `interfaces` are interchangeable. Thus, each `interface` poses a point of variability in its own. Additionally, each module usually has a clearly defined functionality. This functionality is encapsulated by the `interface` of the module and thereby can be reused in another context. If the system is merely a composition of fine granular modules, then a composition of other modules will result in another system with a different functionality as variability is usually nothing more than a tree composed of different modules for the same `interface` and a resulting system is nothing more than a path through such a tree, variability directly results from a modular system.

Third, recertification is simplified if each module is guarded by an `interface`. An `interface` poses a natural barrier for unwanted side-effects of changes. In this way one can prove more easily that if one module is changed, then other, already existing modules are not affected by this change. This way a system designer is only left with the task of recertifying the changed module instead of recertifying the system as a whole.

The rest of this section is structured as follows: section 4.2 presents the adaptations made in contrast to our former approach, as well as our mapping from AADL to OSGi and is subdivided into subsections dealing with different the different aspects of this mapping, i.e., a translation of type and implementation declarations, hierarchies and refinements, **`ports`**, **`threads`**, as well as semantic connections. In subsection 4.3.2 we detail our mapping of AADL **`modes`** onto sets of configurations in OSGi. In section 4.4 we compare our approach to existing approaches in the same field of research. Finally, we evaluate our modified approach by a comparison between handwritten code, generated code from our previous approach and generated code of the current approach.

4.2. Mapping Adaptations

As the mapping between AADL and OSGi is based upon our former work described in chapter 3, we will compare the mapping presented in this chapter to the one defined in section 3.2. In some parts we just reuse the former defined mapping, but in others we need to adapt the mapping in order to be able to generate modular code that adheres to the restrictions of OSGi and also makes use of its best practices.

First, we will present the differences between the two approaches and the adaptations made in our current approach. Afterwards, we will introduce the enhancements provided by the current approach regarding configuration and inter-process communication.

4.2.1. Type and Implementation Declarations

Type declarations in AADL are representing interfaces in form of **`features`** that are provided by a realizing implementation declaration. In the former approach we reused the concept of an `interface` that already exists in Java by translating each **`feature`** - which were restricted to **`data ports`** only - defined by a type declaration into corresponding simple `in` and `out` methods. A drawback of this approach was the fact that **`features`** were regarded as part of a parent component and not as a component of their own. This way a **`feature`** transformed to code could semantically not be altered afterwards. Additionally, the respective parent component relied on an external component that calls the defined methods to transfer data form and to the component instead of a self-contained approach, where each component takes care of sending and receiving on its own.

The mapping presented in this chapter will alter the mapping defined before in order to tackle those drawbacks and by this create a more modular code base for mapped AADL type declarations and **`features`**. In general, we decided to keep the mapping from a type declaration to a Java `interface`, as the abstract nature of an `interface` reflects those of a type declaration. However, instead of mapping each **`feature`** onto a respective method declaration we decided to map each **`feature`** onto a self-contained `class` that is no more reflected in the type declaration `interface`, but will later be referenced in the `class` of an implementing implementation declaration. This will be shown later when we explain the changes made to the mapping regarding implementation declarations.

This mapping of a type declaration **threadA** on its corresponding `interface` is shown in Listing 4.1 and Listing 4.2, as is the mapping from the defined **feature** `dataIO` onto a corresponding Java `class`. Aside from the obvious change from an `interface` method declaration to a self-contained `class`, Listing 4.2 also shows a second change we made in this mapping: the use of OSGi. The generated `class` makes use of the `@Component` annotation of OSGi. As **features** can not have a hierarchy like type declarations or implementation declarations they consequently do not implement any `interface` in their Java representation. However, in order to later reference a **feature** `class` from an implementation declaration `class` it has to define a type it is registered under in the OSGi service registry. This is done via the annotation property `service` in Listing 4.2 which registers `DataIO` directly under its own `class`.

```
thread threadA
  features
    dataIO : in out data port A;
end threadA;

data A
end A;
```

Listing 4.1: Types and features in AADL

```
public interface ThreadA{}

@Component(service = DataIO.class)
public class DataIO{
  public void setData(A data){...}
    public A getData(){...}
}

public interface A{}
```

Listing 4.2: Types and features in Java/OSGi

In order to use Java with OSGi as our target language we have to make sure, that the rules of inheritance posed by AADL on implementation declarations can be reflected by an OSGiified code base. Remember, a implementation declaration can **realize** exactly one type declaration and **extend** exactly one other implementation declaration. Multiple inheritance is forbidden by the standard. The extended implementation declaration has to **realize** the same type declaration as the extending one. One type declaration can have several implementation declarations, each inheriting the **features** defined by the type declaration.

As we changed the way a type declaration and **`features`** are mapped onto Java, we have to adapt the way implementation declarations are mapped too. In our former approach implementation declarations inherited all **`feature`** methods automatically, as they were implementing the type declaration's `interface`. In our current mapping this no longer holds true, as a type declaration `interface` does no longer contain any **`feature`** method declarations. Instead, **`features`** are `classes` of its own now. Therefore, implementation declarations have to somehow reference those `classes`.

In order to adhere to OSGi-specific restrictions regarding inheritance and still generate a semantically compliant code base we used OSGi's `@Reference` mechanisms to reference **`features`** as we formerly referenced **`subcomponents`** via a member variable in an implementation declaration's `class`. Now, each **`feature`** defined by any parent type declaration of an implementation declaration is added to the **`implementation declaration's`** respective `class` as a member variable that is annotated with `@Reference`, which can be seen in Listing 4.3 and Listing 4.4. The `filter` of the `@Reference` can be set to any property that is unique to the instance of the respective **`feature`** `class`.

```
thread implementation threadA.impl
end threadA;
```

Listing 4.3: Implementation declarations in AADL

```
@Component
public class ThreadAImpl implements ThreadA{
    @Reference
    private DataIO dataIO;
}
```

Listing 4.4: Implementation declarations in Java/OSGi

Additionally, implementation declarations can have **`subcomponents`** which specify the inner composition of a component. In the former approach those were represented as member variables that hold references to the respective **`subcomponents`**. Those were set by a constructor that had the same number of parameters as the component had **`subcomponents`**. A drawback of such an approach is again the assumption about an outer component that takes care of assembling the respective component with all its **`subcomponents`**.

In our current approach this is solved by the same means a **`feature`** is implemented in an implementation declaration `class`. **`Subcomponents`** in AADL are composed of a type, usually another implementation declaration and a name for the instance of such. Therefore, the mapping is straight forward and an implementation declaration `class` declares a member variable for each **`subcomponent`** that is defined for the corresponding implementation declaration. In contrast to the former approach this member variable also has a `@Reference` annotation that references the respective instance of the declared type as can be seen in Listing 4.5 and Listing 4.6. Listing 4.5 depicts a **`process implementation`** **`processA.impl`** of the process type **`processA`** which defines two **`thread`** **`subcomponents`**, **`worker1`** and **`worker2`**. This implementation declaration is translated into a Java `class` as shown in Listing 4.6. The `class` `ProcessAImpl` implements the `interface` `ProcessA` and represents each **`subcomponent`** by its own member variable, i.e., `worker1`, `worker2` and uses the declared classifier as type, i.e., `ThreadAImpl`. Formerly, all those member variables had to be initialized within the constructor by an outside entity. This is no longer necessary due to the mechanisms provided by OSGi, i.e., the `@Reference` annotation which enable a programmer to set those references and changing them by setting the configuration of a component. This way the respective instance is set via the target filter that is given by the component's configuration.

```
process processA
end processA;

process implementation processA.impl
  subcomponents
    worker1 : thread threadA.impl;
    worker2 : thread threadA.impl;
end processA;
```

Listing 4.5: Implementation declarations with subcomponents in AADL

```
public interface ProcessA{}

@Component
public class ProcessAImpl implements ProcessA{
    @Reference
    private ThreadA worker1;

    @Reference
    private ThreadA worker2;
}
```

Listing 4.6: Implementation declarations with subcomponents in Java/OSGi

4.2.2. Hierarchies and Refinements

In the former approach we mapped the inheritance semantics of AADL onto Java simply be reusing the `extends` keyword in Java. As the **`extends`** relation of AADL only allows single inheritance and demands the extending component to be of the same type, e.g., **`system`**, **`process`**, etc., as the one that is extended, this was a legitimate approach. This holds true in parts for our current approach, where we made changes to the way **`features`** are mapped to the corresponding Java code. Type declarations inherit all **`features`** from their parent type declaration which formerly was mapped in Java by the child `interface` inheriting all method signatures from the parent `interface`. As we now have no more method signatures to inherit, this boils down to a simple inheritance hierarchy with no methods for type declarations.

The same concept was applied to implementation declarations that are mapped as `classes` instead of `interfaces`. An implementation declaration extending another implementation declaration simply inherited all **`features`** and **`subcomponents`** from its parent which meant that all sub`classes`/-`interfaces` in Java inherited all non-private member variables, representing **`subcomponents`** and all methods representing **`features`**.

First, this no longer holds true for our current approach where the inheritance mechanisms of Java methods cannot be applied to our new `interfaces` that do not contain method declarations any longer. In case of an implementation declaration realizing a type declaration, we still can use the `implements` keyword in Java to map the semantic meaning of a hierarchy, but the methods are no longer there and therefore are not inherited. Each **`feature`** defined in a type declaration has to be implemented as a private member variable in each of the direct children of this type declaration. This way each implementation declaration child redeclares all **`features`** defined by any of its parent type declarations at once.

Second, OSGi poses restrictions regarding the use of DS annotations within an inheritance hierarchy. Given two `classes` A and B and B extends A, then it is not possible to define a `@Reference` annotation on a member variable of A that is then also treated as an OSGi reference in B. OSGi merely treats the `@Reference` annotations of B as references, but not those of its parent. Therefore, we have to repeat this for every child within the inheritance tree. This way each direct and indirect implementation declaration within an inheritance tree redeclares all **`features`** of all its type declaration parents as private member variables with a `@Reference` annotation. As the member variables are declared as private they can have the same name within any `class` in this inheritance tree.

As **`features`** are treated the same way as **`subcomponents`** and both are defined as member variables with a `@Reference` annotation. This has to be repeated for all **`subcomponents`** as well, i.e., all implementation declaration `classes` have to declare a private member variable for each **`subcomponent`** reference defined in any of its parent implementation declaration `classes`. The new approach can be seen in Listing 4.7 and Listing 4.8. Listing 4.7 defines two type declarations, i.e., **`threadA`** and **`threadB`** as well as two corresponding implementation declarations, i.e., **`threadB.impl1`** and **`threadB.impl2`**. **`threadB`** extends **`threadA`** and therefore inherits its **`features`**, i.e., **`dataIO1`**. **`threadB.impl1`** implements **`threadB`** and therefore inherits the **`features`** of both **`threadA`** and **`threadB`**. Finally, **`threadB.impl2 extends threadb.impl1`** thus inheriting all of the **`features`** defined by **`threadA`** and **`threadB`** and additionally all **`subcomponents`** defined in **`threadB.impl1`**. In Listing 4.8 this hierarchy is reflected in code by the use of `extends` and `implements` keywords. However, as **`features`** are no longer represented by method declarations in `interfaces`, they are consequently not automatically inherited by sub`classes` or -`interfaces`. Instead, the corresponding member variables are declared in each sub`class`, adding up in each additional sub`class`, e.g., `ThreadBImpl1` only has member variables for `dataIO1`, `dataIO2` and `sub1`, whereas `ThreadBImpl2` has all of the above mentioned and additionally a member variable for `sub2`.

```
thread threadA
  features
    dataIO1 : in out data port A;
end threadA;

thread threadB extends threadA
  features
    dataIO2 : in out data port A;
end threadB;

thread implementation threadB.impl1
  subcomponents
      sub1 : subprogram subA.impl;
end threadB.impl1

thread implementation threadB.impl2 extends threadB.impl1
  subcomponents
      sub2 : subprogram subA.impl;
end threadB.impl2
```

Listing 4.7: Hierarchies in AADL

```
public interface ThreadA{}
```

```java
public interface ThreadB extends ThreadA{}

@Component
public class ThreadBImpl1 implements ThreadB{
  @Reference
  private DataI01 dataI01;

  @Reference
  private DataI02 dataI02;

  @Reference
  private SubAImpl sub1;
}

@Component
public class ThreadBImpl2 extends ThreadBImpl1{
  @Reference
    private DataI01 dataI01;

    @Reference
    private DataI02 dataI02;

    @Reference
    private SubAImpl sub1;

    @Reference
    private SubAImpl sub2;
}
```

Listing 4.8: Hierarchies in Java

Although this solution is a working one, one might argue that it is not an architectural good one. However, this solution solves a problem that occurred in our first approach, i.e., invalid method signatures when using **`refines to`** declarations in AADL.

In our former approach problems arose when it came down to the refine mechanism in AADL. The refinement of a component encompasses, among other things, the possibility to refine classifiers of **`ports`** by an extending type declaration. In order to reflect the classifier refinement of a **`data port`** in an extending and refining type declaration, we had to change the signature of its corresponding method in Java. In Java, this led to an `interface` definition with an invalid method signature. An overriding method must have the same signature, composed by method name and parameter types, as declared in the super type. As the overriding method changes the parameter types, it is not valid.

In our current approach we do not have to deal with such invalid method signatures anymore, as the type declaration `interfaces` do not contain methods any longer. At the time a **`feature`** is implemented by an implementation declaration the type of the **`feature`** can not change any more and therefore the implementation declaration `class` declares the most "up-to-date" type for the **`feature`** member variable. This is shown in Listing 4.9 and Listing 4.10.

In Listing 4.9 we define two **`thread`** type declarations **`threadA`** and **`threadB`**, where **`threadA`** defines an **`in out data port`** with type **`A`** that is refined to an **`in out data port`** with type `B` within **`threadB`**. The depicted **`thread`** implementation declaration **`threadB.impl`** inherits the defined **`data port`** from **`threadB`** that it is implementing. In Listing 4.10 this constellation is represented in its corresponding Java/OSGi form. The `interfaces` for both type declarations are empty and merely reflect the type hierarchy, whereas the `class ThreadBImpl` declares a member variable for the **`feature`** defined in its parent type declarations. As there are two different **`features`** defined in AADL - one with type **`A`** and one with type **`B`** - two different **`feature`** `classes` are created, i.e., `DataIO` which implements `InOutDataPort<A>` and another `DataIO` that implements `InOutDataPort<B>`. Both `classes` are merely differentiated by their implemented `interface` which is typed differently either with `A` or `B`. The two `interfaces` will be explained in detail in subsection 4.2.4.

```
thread threadA
  features
    dataIO : in out data port A
    {Classifier_Substitution_Rule=>Type_Extension};
end threadA;

thread threadB extends threadA
  features
    dataIO : refined to in out data port B;
end threadB;

thread implementation threadB.impl
end threadB.impl;

data A
end A;

data B extends A
end B;
```

Listing 4.9: Refinement of features in AADL

```
public interface ThreadA{}

public interface ThreadB extends ThreadA{}

@Component
public class ThreadBImpl implements ThreadB{
  @Reference
    private DataIO dataIO;
}

```

```
@Component(service=DataIO.class)
public class DataIO implements InOutDataPort<B>{...}

@Component(service=DataIO.class)
public class DataIO implements InOutDataPort<A>{...}

public interface A{}
public interface B extends A{}
```

Listing 4.10: Refinement of features in Java/OSGi

The additionally depicted **`Classifier_Substitution_Rule`** which restricts the possible types for refinement, is kept as we defined it to be in subsubsection 3.2.1.2.

4.2.3. Separation of User- and Framework-specific Code

In the last subsection the `interface InOutDataPort<type>` has been firstly introduced. This `interface` and many others are part of our newly introduced separation of user- and framework-specific code. In our former approach, a user who used the generated code always had the same view on the code as the framework had, i.e., he used the same `classes` and `interface` as were used by the framework. In our current approach one major contribution is the separation of user- and framework-specific code, which is achieved by the use of user- and framework-specific `interface` for one `class`. This concept was implemented in order to restrict the possibilities of a user who extends the generated code to affect the inner workings of the framework as well as to hide the complexities of the generated framework from the potential user. Classes that implement both types of `interface` are used to provide framework-specific data to the user or the other way around. In Listing 4.11 we depicted an abstract example for such a `class` with user- and framework-specific `interface`. The user-specific `interface IUser` defines two methods in order to set and get a `name` property. The framework-specific `interface` merely has a getter method for a `name` property. Both `interface` are implemented by the `DataClass`, which, for the sake of simplicity, knows about how the framework expects its `name` property, i.e., with a "framework" prefix. The `classes UserSpecificCode` and `FWSpecificCode` represent code written by a potential user and code that is generated by our approach and thus part of the framework. The `class UserSpecificCode` declares a reference to `IUser`, whereas the `class FWSpecificCode` declares a reference to `IFramework`. Both references are satisfied by OSGi with the same service, i.e., the same instance of `DataClass`, as it implements both `interface` and therefore is registered under both in the OSGi registry. Now both parties have access to the same data but with their own special view on it and can communicate with each other without even suspecting the existence of the other.

```java
public interface IUser {
  public String getName();
  public void setName(String name);
}

public interface IFramework {
  public String getNameFW();
}

@Component
class DataClass implements IUser, IFramework {
  private String nameUser = "default";
  private String prefixFW = "framework";
  public String getName() { return nameUser; }
  public void setName(String name) { nameUser = name;}
  public String getNameFW() { return prefixFW + nameUser; }
}

@Component
class UserSpecificCode {
  @Reference
  private IUser data;
  private void doSomething(){
    String name = data.getName();
    ...
    data.setName(name);
  }
}

@Component
class FWSpecificCode {
  @Reference
  private IFramework data;
  private void doSomething(){
    String name = data.getNameFW();
    ...
  }
}
```

Listing 4.11: User- and framework-specific interfaces and data classes

This kind of communication between framework- and user-specific code is used in different parts of our mapping, e.g., for **`threads`** as explained in subsection 4.2.5 and ports as explained in subsection 4.2.4. The existence of different views, i.e., `interfaces`, will be mentioned if sensible, but will be left out for the sake of brevity where possible in our further work.

4.2.4. Ports

As already indicated in subsection 4.2.2 **`features`**, in particular **`data ports`**, have been completely redesigned in our current approach. Instead of a direct mapping onto methods of the type declaration `interface` those **`features`** belong to, they are now mapped onto self-contained **`feature`** `classes` that are referenced via member variables in the corresponding implementation type `classes` and OSGi's `@Reference` annotations. This new concept was already depicted in Listing 4.1 and Listing 4.2. Also the separation of user- and framework-specific code that leads to the use of special `interfaces` has been mentioned in subsection 4.2.3. However, the implementation details of the respective ports have been left unclear.

In this subsection we will restrict ourselves to **`data ports`** as those are the only ones supported currently by our approach. First we will examine how plain AADL defines the semantics of **`data ports`** and their methods and then we will explain the additional semantic restrictions that we introduced with respect to how users of the generated framework shall interact with them.

AADL defines the semantics for **`data ports`**, as well as methods they have to implement in order to interact with them, e.g., the definition given by AADL for **`Receive_Input`** defines its semantic as

> "[...] explicitly request port input on its incoming ports to be frozen and made accessible through the port variables, any previous queue content not processed by `Next_Value` calls is discarded. Newly arriving data may be queued, but does not affect the input that thread has access to." [18]

This definition encompasses semantics not only for **`data ports`**, but also all other kinds of **`ports`**. Therefore, some of its definitions do not match **`data ports`** perfectly, e.g., a **`Next_Value`** method would make no sense on a **`data port`** as it is defined to only hold one data value at a given point in time and not a queue of data values. Thus, we discard the parts of the definition not applying to **`data ports`** and reduced the set of applicable methods to **`Receive_Input`**, **`Get_Value`**, **`Set_Value`** and **`Send_Ouput`**.

The definitions given by AADL for those methods are:

> "[...] `Get_Value` to access the current value of a port variable. Re-

> peated calls to Get_Value result in the same value to be returned, unless the current value is updated through a Receive_Input call or a Next_Value call." [18]
>
> "Put_Value: supply a data to a port variable." [18]
>
> "Send_Ouput: explicitly cause event, event data, or data to be transmitted." [18]

In our former approach those semantics were not considered in detail, but merely replaced by simple `getter` and `setter` methods. In our current approach we extend the mapping in order to adhere to the semantics given by the AADL standard in Java/OSGi. We ensure that each **`data port`** `class` implements the respective methods, i.e., **`Receive_Input`** and **`Get_Value`** for **`in data ports`**, **`Set_Value`** and **`Send_Ouput`** for **`out data ports`** and all of them for **`in out data ports`**.

However, as explained in subsection 4.2.3 our approach now differentiates between framework- and user-specific code, so we added additional restrictions to the **`data ports`** that are used by users.

First, we assume, that all timing-related semantics are defined by the AADL model, e.g., when data of a **`data port`** is sent or when it is received in relation to their parent component's lifecycle methods. Therefore, on user-side there is no need for an explicit **`Send_Output`** or an explicit **`Receive_Input`** method. Those semantics are handled by the framework, which takes care of when data is received, when frozen and made available to the user and when it is sent to other components in the framework. This reduces the set of methods needed on user-side to **`Get_Value`** and **`Put_Value`**, whereas on framework-side the set of methods encompasses all methods, but **`in data ports`** and **`in data ports`** have another semantic, i.e., to retrieve the value set by a user and to set the value the user should interact with in the next lifecycle of the respective component.

In the exemplary case of an **`in out data port`** this results in two different `interfaces` as depicted in Listing 4.12. The `interface IInOutDataPort` reflects the user-side by only declaring the two user-side required methods `getValue` and `putValue`. The `interface IFWInOUtDataPort` represents the framework-side by declaring the framework-specific `getValueFW` and `putValueFW` methods that have a different semantic than their user-side counterpart as described before. Additionally, the methods `receiveInput` and `sendOutput` are declared to be used by the framework to send and receive data on/over this port at the predefined times in the AADL model. Excluding the framework-specific methods with an FW suffix, those methods represent those required by AADL for all **`data ports`**.

```
public interface IInOutDataPort<T>{
  public T getValue();
  public void putValue(T data);
}

public interface IFWInOutDataPort<T> {
  public void receiveInput();
  public Integer getValueFW();
  public void putValueFW(T data);
  public void sendOutput();
}
```

Listing 4.12: User and framework interfaces for an in out data port

In Listing 4.13 we depicted all method implementations for a sample `InOutDataPort` that is defined with an `Integer` as data type. The stipulated semantics for **`Receive_Input`**, which is only called by the framework, are fulfilled by separating the `Integer` value into two different member variables. One for data to be set during a dispatch, i.e., `inValue`, and one for the port to deliver during this dispatch when `getValue` is invoked, i.e., `frozenInValue`. The value of `inValue` is frozen as soon as a component invokes `receiveInput` by copying the current value of `inValue` to `frozenInValue`. This depicted code is thread-safe as the variables are all declared `volatile`, therefore no write-invalidations can occur if several threads access the variables through the declared methods at the same time. `putValue` is realized as a simple setter method with a volatile, backing member variable `outValue`. `sendValue` is intentionally left blank as the logic of this method is connected to the concept of semantic connections which will be explained later in subsection 4.2.6. In addition, the two framework-specific methods `putValueFW` and `getValueFW` are implemented. The first in order to enable the framework to set the initial `inValue` that is subsequently frozen via `receiveInput` and made available to the user via `getValue`. The second in order to enable the framework to take care of transmitting the data set by a user to by first retrieving this data via `getValueFW` and then subsequently by sending the data via `sendOutput`.

```
@Component
public class InOutDataPort implements IInOutPort<Integer>, IFWInOutPort<
    Integer>{
    private volatile Integer inValue;
    private volatile Integer frozenInValue;
    private volatile Integer outValue;

    public Integer getValue(){return frozenInValue;}
    public void putValue(Integer data){outValue = data;}
    public Integer getValueFW(){return outValue;}
    public void putValueFW(Integer data){inValue = data;}
    public void receiveInput(){frozenInValue = inValue;}
    public void sendOutput(...){...}
}
```

Listing 4.13: Port methods in Java/OSGi

4.2.5. Threads

The mapping between AADL and Java/OSGi has made changes to the aforementioned approach in two ways, but largely stays the same as already explained in subsubsection 3.2.2.1. We keep the translation into a `BoundAsyncEventHandler`, but do not further regard a translation into an `AsyncEventHandler` as sensible. Due to the configuration possibilities of our new approach as later on described in section 4.3. We might otherwise have to change the type of our thread implementation from an `AsyncEventHandler` into a `BoundAsyncEventHandler`, or vice versa, as **`immediate connections`** come and go, which is technically not possible in RTSJ. The mapping of the **`thread's`** properties, i.e., **`Compute_Deadline`**, **`Period`**, **`Priority`** and **`Dispatch_Protocol`**, onto the corresponding Java constructs, i.e., `DeadlineMissHandler`, `ReleaseParameters` and `PriorityParameters` was kept.

The first major change concerns the concept of the central piece of code, i.e., a `Main class` with a `main` method as described in subsection 3.2.4, that starts all threads of a generated system from the outside. This contradicts our concept of modularity, where each component is self-contained and, in case of a **`thread`**, knows when to start and stop itself. Therefore, we made use of the lifecycle methods of immediate and delayed services in OSGi that have been introduced in subsubsection 2.4.3.2. As described in subsection 4.2.1 all AADL components are translated into `classes` with an OSGi `@Component` annotation, which makes them either an immediate or a delayed component. Thus, threads are OSGi components that can make use of OSGi's `@Activate`, `@Deactivate` and `@Modified` annotations to participate in the lifecycle of a component. We use those annotations to annotate methods that implement the starting and stopping of the thread. This can be seen in Listing 4.14 where we defined a **`thread`** implementation declaration **`ThreadC.impl`** that defines several properties altering its semantic behavior. First, the **`Dispatch_Protocol`** is set to **`periodic`**, which makes this thread execute in a fixed interval given by the **`Period`** property of 200ms. **`Priority`** is set to 5 and the **`Compute_Deadline`**, i.e., the time until which the thread has to finish its work, is set to 100ms relative to its dispatch. Those declarations are mapped to Java/OSGi as depicted in Listing 4.15 where the periodic nature of `TreadCImpl` is reflected by the use of a `PeriodicTimer` whose period is set to 200ms. The priority is set to 5 within the given `PriorityParamters` and the `ReleaseParamters` define a deadline of 100ms. Instead of declaring public `start` and `stop` methods to be called manually by an outside entity, `ThreadCImpl` makes use of OSGi's lifecycle annotation to mark the methods `initialize` and `finalize` to be called when the service is started, respectively stopped by OSGi. This way we create **`threads`** that are self-contained in contrast to **`threads`** in our former approach, where an outside entity had to start and stop the thread.

```
thread ThreadC
end ThreadC;

thread implementation ThreadC.impl
  properties
    Dispatch_Protocol => periodic;
    Period => 200ms;
    Priority => 5;
    Compute_Deadline => 100ms;
end thread.impl;
```

Listing 4.14: Thread in AADL

```
@Component
public class ThreadCImpl implements ThreadC{
  public AsyncEventHandler handler;
  public Timer timer;

    @Activate
    public void initialize() {
        timer = new PeriodicTimer(null, new RelativeTime(200, 0), null);
        ReleaseParameters rps = timer.createReleaseParameters();
        handler = new BoundAsyncEventHandler(){
            public void handleAsyncEvent(){
              dispatch();
                start();
                compute();
                completion();
            }
        };
        handler.setDaemon(false);
        handler.setSchedulingParameters(new PriorityParameters(5));
        rps.setDeadline(new RelativeTime(100, 0));
        handler.setReleaseParameters(rps);
        timer.addHandler(handler);
        timer.start();
    }
    @Deactivate
    public void finalize() {
      timer.stop();
    }

    ...
}
```

Listing 4.15: Reuse of lifecycle methods in Java/OSGi (Simplified)

Listing 4.14 is marked as being a simplified version of an actual **`thread`** implementation. This is due to the second major enhancement we made.

In AADL **`threads`** are defined to have several lifecycle events and states, e.g., **`Initialize`**, **`finalize`**, **`activate`** or **`compute`**. Each of them can be defined by a predefined **`Programming_Property`** that is named accordingly, e.g., **`Compute_-Entrypoint`** or **`Activate_Entrypoint`**. The declaration of such a property can be seen in Listing 4.16. The value of each of those properties is a reference to a **`Subprogram`** that shall be executed in case the respective event is triggered. A **`subprogram`** in AADL can have parameters and resembles a method in Java. **`Subprograms`** that are chosen to be the method to execute for a given **`thread`** lifecycle event are defined to have access to all variables that the **`thread`** itself has access to, which in our case are usually all its **`data ports`** and the `BoundAsyncEventHandler` and its `Timer`.

```
thread implementation ThreadA.impl
  properties:
      Activate_Entrypoint => InitializeThreadA.impl;
end ThreadA.impl;

subprogram implementation InitializeThreadA.impl
end InitializeThreadA.impl;
```

Listing 4.16: Declaration of an Activate_Entrypoint

In our previous approach we decided to simplify the lifecycle events of a **`thread`** to be represented by several lifecycle methods, e.g., `activate`, `deactivate` or `compute`. Due to the modularity we had in mind for our current approach, this no longer fits our needs. Once the methods are generated, they can not be altered anymore which makes them neither configurable nor exchangeable. Therefore, we decided to externalized the logic that usually is called during lifecycle methods of a thread, e.g., initialize or finalize, but also when switching between states or, although not included in this work, when reacting to events. The logic formerly contained within a method now is moved into a separate `class` that represents the **`subprogram`** executed in a thread-specific context.

As the respective **subprogram** must be granted access to all the variables available to the context they are used in, i.e., in this case all the variables that are also available to `ThreadCImpl`, we pass this context as a generic `Thread` parameter that has then to be cast into the respective implementation that is the context of the current **subprogram**. An example implementation for the activation of `ThreadCImpl` is shown in Listing 4.17.

```java
@Component
public class ThreadCImpl implements ThreadC{
  public AsyncEventHandler handler;
  public Timer timer;

    @Reference
    Subprogram initialize;

    @Activate
    public void initialize() { ... initialize.execute(this); ... }
    ...
}

@Component
public class InitializeThreadAImpl implements Subprogram{
  public void execute(Thread context) {
    ThreadCImpl myContext = (ThreadCImpl) context;
    myContext.timer = new PeriodicTimer(null, new RelativeTime(200, 0), null);
    ReleaseParameters rps = timer.createReleaseParameters();
    myContext.handler = new BoundAsyncEventHandler(){
    public void handleAsyncEvent(){
        myContext.dispatch();
        myContext.start();
        myContext.compute();
        myContext.completion();
      }
    };
    myContext.handler.setDaemon(false);
    myContext.handler.setSchedulingParameters(new PriorityParameters(5));
    rps.setDeadline(new RelativeTime(100, 0));
    myContext.handler.setReleaseParameters(rps);
    myContext.timer.addHandler(handler);
    myContext.timer.start();
  }
}
```

Listing 4.17: The generic parameter of a subprogram

`InitializeThreadAImpl` declares a method `execute` that can be called by its context, i.e., `ThreadCImpl`, in order to execute the respective action, e.g., `compute` or `activate`. Within the `execute` method the generic `context` parameter then is cast into the actual type, i.e., `ThreadCImpl`. Afterwards, the method executes the same code as the `activate` method of the **thread** has done before by using the variables of the respective `context` object. This way we decouple the logic from the **thread** and by using OSGi's referencing mechanisms the method is exchangeable at runtime.

4.2.6. Semantic Connections

In subsection 3.2.3 of our former approach we explained how the communication-related mapping between AADL components and RTSJ was handled. In order to enable all components to communicate via their defined **`port connections`** and to hide those mechanisms from the user of the generated framework, we introduced an extra `class` for each component, called a CB. This `class` took care of transmitting data from one **`port`** to another as long as both **`ports`**, source and target, were defined within the component the CB belonged to. Thus, the concept of **`connections`** had not been translated into a separate `class` for each **`connection`**, but into one `class` comprising all **`connections`** defined within one component.

This again contradicts our understanding of modularity and also impedes the configurability of a component where **`connections`** can come and go depending on the current state of a component, or can be switched from being immediate to being delayed during runtime. Additionally the logic of how data is transmitted between **`ports`** is located within an outside entity instead of the **`ports`** an **`connections`** itself, what would be preferred in a self-contained design. Therefore, in our current approach we decided to represent not only **`ports`** as OSGi components, but **`connections`** as well. An example of such a **`connection`** in AADL and its representation in OSGi is given in Listing 4.18 and Listing 4.19.

In Listing 4.18 a **`process`** implementation **`ProcessA.impl`** is declared to have two **`subcomponents`** **`threadA`** and **`threadB`**. In our example those are assumed to each have a **`data port`** defined, **`threadA`** an outgoing and **`threadB`** an incoming one. Both **`ports`** are then declared to be connected via a **`port connection`** `con1`. In Listing 4.19 this **`connection`** is represented in OSGi via the component `Con1` implementing the generic `interface` `PortConnection<T>`. In this case we assumed each **`data port`** to have its data type to be declared as `Integer`. In case the two ports had different types declared which is possible due to the **`Type_Extension`** value of the **`Classifier_Matching_Rule`** as described in subsubsection 3.2.3.1, then the **`connection`** would have the common super type of both classifiers as data type. We only consider directed **`connections`**, meaning that a **`connection`** always has a source and a target, represented in our example by OSGi references on an `IFWOutport source` and an `IFWInPort target`. `Con1` itself barely implements any logic but to transmit data from its source to its target.

```
process implementation ProcessA.impl
```

```
  subcomponents
      threadA: thread ThreadA.impl;
      threadb: thread ThreadB.impl;
    connections
      con1: port threadA.out -> threadB.in;
end ProcessA.impl;
```

Listing 4.18: Declaration of a connection in AADL

```
@Component
public class Con1 implements PortConnection<Integer>{
    @Reference
    IFWOutPort<Integer> source;

    @Reference
    IFWInPort<Integer> target;

    public void transmit(){ target.putValueFW(source.getValueFW()); }
}
```

Listing 4.19: Declaration of a connection in Java/OSGi

Based on **`threads`**, **`ports`** and **`connections`** being represented as OSGi components we are now able to express the concept of a semantic connection, as already mentioned in subsection 3.4.2, more accurately than before.

In our previous approach semantic connections only were created implicitly by consecutively calling several CB in a row which eventually forwarded data from an ultimate source to an ultimate target **`port`**, whereas each CB only knew one step, i.e., one **`connection`**, of the overall semantic connection.

In order to illustrate this situation we depicted a semantic connection in Figure 4.1. In this figure there are three **`connections`**, i.e., from **`threadA.dataOut`** to **`proc1.comm`**, from **`proc1.comm`** to **`proc2.comm`** and finally from **`proc2.comm`** to **`threadB.dataIn`**. Those three **`connections`** together, as well as their connected **`ports`** represent one semantic connection with the ultimate source **`dataOut`** and the ultimate target **`dataIn`**. In our former approach **`proc1`**, **`SystemA_impl_Instance`** and **`proc2`** each would have had an own CB that takes care of the connection defined for each component, e.g., **`proc1's`** CB would only take care of forwarding data between **`threadA.dataOut`** and **`proc1.comm`**, but not for the other **`connections`**. The semantic connection then was implicitly created by first calling the CB of **`proc1`** which delegated to the CB of **`SystemA_impl_Instance`** that delegated further to the CB of **`proc2`**.

A problem of this approach is its non-configurability as the transmissions are hard coded cases within the `transmit` method of a CB. Once generated it is not possible to create new semantic connections or to alter the behavior of an existing one during runtime, e.g., due to the change from being an immediate to being a delayed **`connection`**.

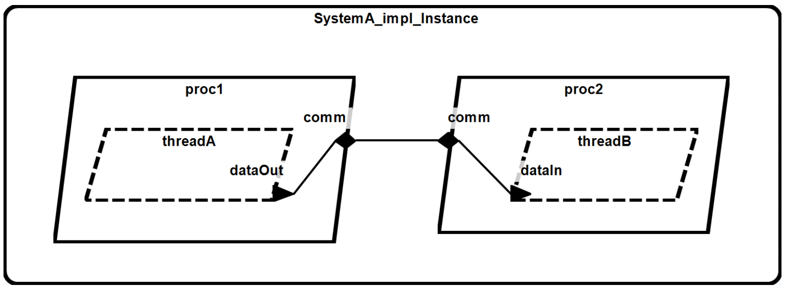

Figure 4.1.: AADL example for a semantic connection

Therefore, we altered our former approach by generating OSGi components that represent semantic connections by referencing all parts a semantic connection consists of, i.e., **`ports`**, **`connections`** and other components guaranteeing their semantic compliance to their AADL counterpart. In Listing 4.20 we depicted a representation of the semantic connection presented in Figure 4.1 as an OSGi component.

```
@Component
public class SemCon1 implements SemanticConnection<Integer>{
    @Reference
    IFWOutPort<Integer> dataOut;
    @Reference
    IFWInPort<Integer> dataIn;
    @Reference
    IFWInOutPort<Integer> comm1;
    ...
    @Reference
    PortConnection<Integer> con1;
    @Reference
    PortConnection<Integer> con2;
    ...
    @Reference
    IFWPhaser phaser;
}
```

Listing 4.20: Declaration of a semantic connection in Java/OSGi

`SemCon1` defines several references to other OSGi components that represent the Java equivalents of AADL **`data ports`** and **`connections`**, e.g., `dataOut` and `dataIn` as ultimate target and source as well as intermediate **`ports`** like `comm`. **`connections`** are references as well as depicted by the references `con1` and `con2`. This way all necessary parts of the semantic connection depicted in Figure 4.1 are reflected as references in its OSGi counterpart `SemCon1`.

The concept of a semantic connection consisting of several real **`connections`** is necessary as only a semantic connection knows if it is immediate, delayed or sampled. Those three possible values of the ***Timing*** property, as described in subsubsection 2.2.4.3, can only be applied to **`connections`** and not semantic connections, as those are completely implicit. Given the case of three **`connections`** connected through **`ports`**, as shown in Figure 4.1, the middle one might be declared to be immediate, whereas the first and last one have no explicit **`Timing`** property set and are therefore sampled by default. AADL defines a semantic connection to be immediate or delayed as soon as one **`connection`**, that is part of the semantic connection, is declared immediate or delayed. Without the notion of a semantic connection as defined in Listing 4.20 the first and last **`connection`** in our example would not know that they are implicitly immediate or delayed as they know nothing of the other **`connections`** they are connected to through intermediate **`ports`**. However, a semantic connection can check each **`connection`** it consists of being immediate or delayed and if so, changing the way data is sent to the ultimate target, depending on the respective semantics of either immediate or delayed.

This is where the above depicted `IFWPhaser` comes into play. This reference is only used in a scenario where one of the referenced **`connections`** is declared to be immediate which forces the semantic connection to somehow force the receiving **`thread`** to run directly after data has been send by the sending **`thread`**. Therefore, each OSGi representation of an AADL **`thread`** has a reference to an `IFWPhaser` that is used as a barrier as soon as an immediate semantic connection is attached to this **`thread`**. Both, **`thread`** and semantic connection, reference the same `IFWPhaser` which they use to synchronize on. As long as the semantic connection did not arrive at the `IFWPhaser` the **`thread`** halts its execution and waits for the sender to send its data over the semantic connection. As soon as the semantic connection arrives at the `IFWPhaser` the **`thread`** resumes its execution and can use the transmitted data during its dispatch cycle.

A semantic connection is only meant for use by the generated framework code. This is reflected in the aforementioned user-specific `interfaces` for **`ports`**, which are the only way for a user to send and receive data within a **`subprogam`**. The actual transmission of data is predefined by the timing semantics of the **`port connections`** and therefore is part of the generated framework. The framework again uses the semantic connections instead of directly calling the `sendData` methods of **`ports`** to transmit data from source to target. This is done in order to encapsulate the knowledge of **`connections`** being immediate, delayed or sampled within those instead of, as before, in an external `class` like the CB.

All of the aforementioned adaptations made to our former approach serve an increased modularity of the overall system. Our final goal is to make the whole system runtime reconfigurable, so that each single part can be reused in other contexts as well. In order to achieve this goal we made heavy use of OSGi components and references. What we omitted until now are the details on how each component gets the right reference and not an arbitrary one. For this we will make use of OSGi's configuration mechanisms which will be explained in detail in the next section.

4.3. Configuration

Based on the changes presented in section 4.2 that were made to our former approach presented in section 3.2, we now explain how OSGi's configuration mechanisms can be leveraged to turn the aforementioned components and their references into a runtime reconfigurable system. Thereby granting enhanced maintainability as each component is self-contained and therefore changeable as long as its contract, i.e., its `interface` is not altered. Additionally, the configurability enables us to map AADL's state mechanisms onto configurations in OSGi.

In the following subsections we first introduce the concepts of properties and target filters in OSGi and how those are connected to configurations in OSGi, followed by a more detailed explanation of states in AADL. Both are connected in the following subsection in order to map AADL states onto OSGi configurations.

4.3.1. Properties and Target filter

In order to understand the concept presented in subsection 4.3.5 we have to revisit the mechanisms OSGi provides with regard to configuration of runtime systems. Those mechanisms can be separated into two distinct categories, i.e., properties/targeting and configurations. In this subsection we explain how OSGi makes objects configurable through properties that are assigned to them which then can be used by a target property for selection of specific instances.

In OSGi each component or service is registered at a central service registry as already explained in subsubsection 2.4.3.3. Additionally, each registered service is accompanied by a `Map` of properties that are specific for this object instance. Those properties can be separated into two types. First, OSGi internal properties that are assigned by the framework and depend on the properties of a given `@Component` annotation. Second, properties that are set by the developer which can be arbitrary. In Listing 4.21 we depicted a component with additional properties set in its `@Component` annotation, i.e., `name`, `service`, `immediate` and `property`. `immediate` and `service` are properties that are used by a Service Component Runtime (SCR) in order to determine under which `interface` it should register a given component and if this component should be started immediately or delayed. `name` is used to set the name of this component which by default is its fully qualified `class` name, i.e., de.unia.smds.osgi.MyComponent. Finally, `property` is used to set arbitrary, user-defined properties for this component. Those properties are always in the format <identifier>:<type>=<value>, where <type> is optional.

```java
@Component(
   name="compName",
   immediate=true,
   service=MyComponent.class,
   property={"ref.target=(sometarget=*)", "ref.cardinality.minimum=1"}
   )
public class MyComponent{
    @Reference(cardinality=OPTIONAL)
    volatile ISomeService ref;

    @Activate
    private void activate(Map<String, Object> props){ ... }
}
```

Listing 4.21: Declaration of a component with specific properties

Partially, the aforementioned properties can be used at runtime as shown by the `activate` method in Listing 4.21. The properties of this component are passed as a `Map<String, Object>` parameter to the activate method by the SCR when it activates this component. Per default only two properties are contained within this map - `component.name` and `component.id`. `component.name` has been set by us to the value `compName`, `component.id` has been left untouched. Those two properties are set by the SCR regardless whether they have been explicitly defined by the developer or not. In contrast, all properties defined in `property` are only passed if they have been defined by a developer.

`component.name` and `component.id` barely influence the behavior of `MyComponent`, whereas the exemplarily set properties `ref.target` and `ref.cardinality.minimum` directly influence the wiring of `MyComponent` to other services. In Listing 4.21 `MyComponent` defines a reference to another service of type `ISomeService`. This reference is per default `static` and any registered service that implements `ISomeService` is accepted. Its optionality is defined to be optional (0..1). The last two characteristics of this reference are altered by the properties added to `MyComponent` via the `property` attribute.

`ref.target` restricts the acceptable services from any component that implements `ISomeService` to components that implement `ISomeService` and have an additional `sometarget` property defined which means that of the two depicted components in Listing 4.22 only `SomeServiceImpl2` is an eligible target for this reference. It defines a property `sometarget` with an arbitrary value, whereas

SomeServiceImpl1 does not provide this property at all, thus being removed from the list of possible targets for the ref reference in Listing 4.21.

ref.cardinality.minimum is a property that, applied to a component, alters the cardinality of a reference of this component. The respective reference is identified by its name which is the name of the variable, i.e., ref, and the value of the property specifies the optionality part of the reference's cardinality, i.e., mandatory. The minimum cardinality of a reference cannot exceed the multiplicity, the second part of the cardinality. The minimum cardinality property can be used to raise the minimum cardinality of a reference from its initial value, e.g., an optional (0..1) cardinality can be raised to a mandatory (1..1) cardinality by setting this property to 1. The minimum cardinality of a reference cannot be lowered. That is, a 1..1 or 1..n cardinality cannot be lowered to a 0..1 or 0..n cardinality because the component was coded to expect at least one bound service. In our example depicted in Listing 4.21 we initially defined the reference to be optional, but via ref.cardinality.minimum we explicitly set its cardinality to be 1 which makes this reference effectively a mandatory one.

```
@Component
public class SomeServiceImpl1 implements ISomeService{ ... }

@Component(property="sometarget=somevalue")
public class SomeServiceImpl2 implements ISomeService{ ... }
```

Listing 4.22: Declaration of two ISomeService implementations

4.3.2. Configurations in OSGi

In the previous subsection we explained how each component in OSGi comes along with a Map of properties, that further describe their behavior at runtime, e.g., via ref.cardinality.minimum. In this subsection we will explain how those properties correlate to configurations in OSGi and how they can be used by us to express different states of a system.

The aforementioned properties of each component have been implicit, that means they are present even if a developer did not define them explicitly, e.g., component-.name. This also means each component is eligible to run, even if some of its properties have not been set explicitly. This is made possible by sensible default values for all properties. In contrast to this default behavior, a developer can also mark a component to only be runable if there is a configuration available for the component at runtime. A configuration is a collection of properties explicitly set for this component which is subsequently passed during activation to the component through its `@Activate` method.

In Listing 4.23 we depicted a component that explicitly requires a configuration for startup. The component itself declares the attributes `configurationPolicy` and `configurationPid` to force the SCR to take care of activating this component only in case a valid configuration for the `configurationPid` *myPid.test* is available at runtime. This configuration is passed to the `activate` method as usual properties. As the `class RequireConfig` knows which properties it provides to be configured, those can then be taken from the `config Map` and used further to execute runtime-specific logic according to the configuration.

```
@Component(configurationPolicy=REQUIRE, configurationPid="myPid.test")
public class RequiresConfig {
    @Activate
    private void activate(Map<String, Object> config){
        Object configObject = config.get("exampleProp");
        // do something with configObject
    }
}
```

Listing 4.23: Declaration of component requiring a configuration

In order to create such a required configuration we can make use of the predefined `ConfigurationAdmin` service which is standardized by the OSGi compendium specification. In Listing 4.24 we depicted the programmatic creation of a configuration via `ConfigurationAdmin`. The `class MyConfigurator` is itself an immediate component that takes care of deploying a configuration for the configruation pid `myPid.test` into the OSGi runtime. This is done via referencing a `ConfigurationAdmin` which is then used to create a `configuration` object for the given configuration pid. The `configuration` expects a `Dictionary<String, ?>` that contains the properties that shall be passed to the configured component. Those are defined via the `Hashtable ht` that contains the `exampleProp` required by `RequiresConfig` to work properly. Finally, the `configuration` is updated which forces the `ConfigurationAdmin` to deploy the new configuration into the runtime. This in turn activates the `RequiresConfig` component and passes the configuration as a `Map` into its `activate` method. A great benefit of this mechanism is its loose coupling between the component providing a configuration and the component requiring the configuration. Additionally, the mechanism itself frees a developer from manually passing a configuration to a component and taking care of instantiating and managing it.

```
@Component
public class MyConfigurator {
    @Reference
    ConfigurationAdmin admin;

    @Activate
    private void activate(){
        Configuration configuration = admin.getConfiguration("myPid.test", "?"
            );
        Hashtable<String, ?> ht = new Hashtable<>();
        ht.put( "exampleProp", new Object());
        configuration.update(ht);
    }
}
```

Listing 4.24: Programmatic creation of a configuration

In combination with properties and target filter as explained in subsection 4.3.1, we now can turn our formerly static system into a runtime configurable one. This is possible due to the fact that properties like `ref.target` and `ref.cardinality.minimum` can be altered by configurations that are dynamically deployed into the runtime via `ConfigurationAdmin`. The startup of a component is guaranteed to wait for the first configuration to be deployed by a central steering component that has no knowledge of any of the components it controls, but merely knows a set of configurations. In Listing 4.25 we exemplarily depict the **`connection`** presented in Listing 4.19 enhanced with properties, target filters and an external `Configurator` providing the initial configuration for this **`connection`**. `Con1` was defined to be the **`connection`** between the two **`data ports`** **`threadA.dataOut`** and **`proc1.comm`** in Figure 4.1. Both **`data ports`**, as well as all other components, are assumed to have a unique id which serves as a target property for the reference target filter for the `source` and `target` references of `Con1`. In our case this unique identifier is depicted exemplarily for `Con1`, i.e., `de.smds.unia.uid`. Those target filters are not set during compile time via the `@Component` annotation and its `property` attribute, as it was shown in Listing 4.21, but are dynamically set by an external component `MyConfigurator` during runtime.

```
@Component(
    configurationPolicy=REQUIRE,
    configurationPid="systemA_impl_instance.proc1.Con1",
    property={"de.smds.unia.uid=systemA_impl_instance.proc1.Con1"}
    )
public class Con1 implements PortConnection<Integer>{
    @Reference
    IFWOutPort<Integer> source;

    @Reference
    IFWInPort<Integer> target;

    public void transmit(){ target.putValueFW(source.getValueFW()); }
}

@Component
public class MyConfigurator {
    ...
    @Activate
    private void activate(){
        Configuration configuration = admin.getConfiguration("
            systemA_impl_instance.proc1.Con1", "?" );
        Hashtable<String, ?> ht = new Hashtable<>();
        ht.put( "source.target", "(de.unia.smds.uid=systemA_impl_instance.
            proc1.threadA.DataOut)");
        ht.put( "target.target", "(de.unia.smds.uid=systemA_impl_instance.
            proc1.Comm)");
        configuration.update(ht);
    }
}
```

Listing 4.25: Programmatic configuration of Con1

There are two fields in which the aforementioned mechanisms of properties, target filtering and configurations enhance our former approach. First, during system startup as will be explained in the following subsection. Second, by enabling us to map AADL states of systems on sets of configurations as we will explain in subsection 4.3.4 and subsection 4.3.5.

4.3.3. System Setup

The system setup described in subsection 3.2.4 encompasses a severe drawback regarding its inability to enable changes or reconfigurations of a system at runtime.

In our former approach there are virtually no means of changing anything in the running system once it has been started which poses a severe restriction to the expressiveness of this approach. In real-world systems often the system under development has several states, e.g., an autopilot being in flying or in landing mode.

Also the need for an external entity, i.e., the `Main class`, can be regarded as a design flaw from the point of view of a truly modular system. A better approach would be to let each part of a system know its direct dependencies and the overall system then results from the implicit transitive dependencies between all of its parts. This would eliminate the need for a central entity with knowledge about how the overall system has to be composed.

Therefore, we dropped the former approach of an external `Main class` with a hard coded `main` method in favor of OSGi's more elegant approach of configurable components, as described in subsection 4.3.1 and subsection 4.3.2.

In order to highlight the differences between our former and current approach we reuse the `AutopilotProcess.impl` presented in section 2.1 as an example system whose startup code shall be generated either as pure RTSJ or RTSJ with OSGi. As a refresher, the system architecture is once again depicted in Figure 4.2.

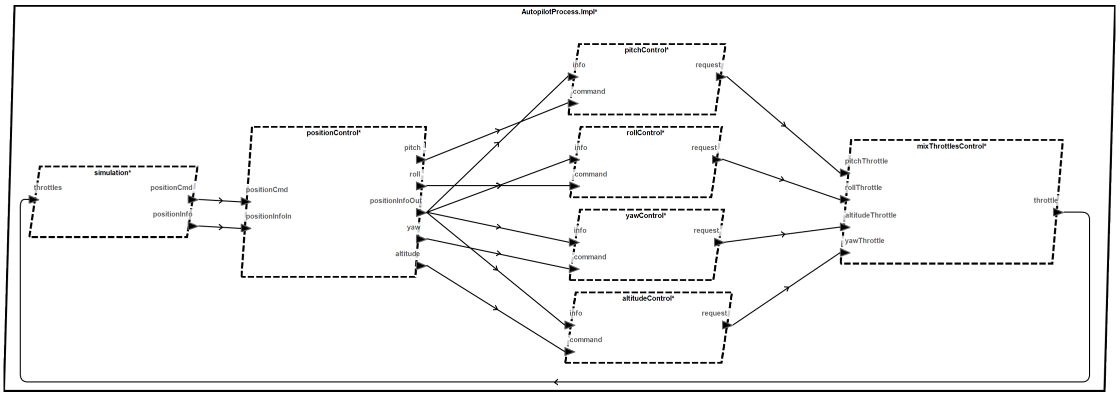

Figure 4.2.: Architecture of the autopilot in AADL

First we will have a closer look at the startup code generated in RTSJ which partially has already been described in subsection 3.2.4. In Listing 4.26 the whole startup code for the system depicted in Figure 4.2 is shown. As all AADL components are represented by `classes` that each know their **`subcompnents`**, only the root components, i.e., the process and its threads have to be created, initialized and started, as well as the CB. This is a straight forward process, but obviously makes it impossible to alter the composition of each of the **`subcomponents`** as well of the threads itself or the process. Each `class` explicitly states which **`subcomponent`** it expects to be passed through its constructor and therefore leaves no space for change.

```
public class Main{
    public static void main(String[] args) {
    instance.autopilotsystem_impl.autopilot.simulation.simulation simulation =
         new instance.autopilotsystem_impl.autopilot.simulation.simulation();
    instance.autopilotsystem_impl.autopilot.rollcontrol.rollControl
        rollControl = new instance.autopilotsystem_impl.autopilot.rollcontrol.
        rollControl();
        ...
    instance.autopilotsystem_impl.autopilot.autopilot autopilot_ID615661078 =
        new instance.autopilotsystem_impl.autopilot.autopilot(simulation,
        rollControl, ...);
        ...
        AutopilotSystem_Impl_Instance_ID1055526063.setParentConnectionBroker(
            null);

    simulation_ID2061775037.startExecution();
    rollControl_ID1288843078.startExecution();
        ...
    }
}
```

Listing 4.26: Programmatic startup of AutopilotProcessImpl

In contrast we depicted a startup configuration for the system generated for an OSGi target. In this case the `Main class` is an OSGi component that references the `ConfigurationAdmin` and uses configurations to setup the components as described in subsection 4.3.2. Therefore each component, e.g., **`process`** or **`thread`**, gets a configuration created by the `Main class`, e.g., `processConfig` or `simulationConfig`. These configurations set the target properties of all references of the respective component. For example, the overall autopilot process `autopilot` has references to all its **`subcomponents`**, e.g., `simulation` or `rollControl`. Those references are set to specific targets via `autopilot`'s reference target properties, e.g., `simulation.target` is set to target `autopilotsystem_impl.autopilot.simulation.simulation`. This is repeatedly done for all components and finally all component configurations are updated and thus deployed to the runtime where the SCR picks up these configurations and instantiates all the respective components accordingly. Therefore, the whole system setup is merely a set of configurations, specific to the respective, initial state of an AADL system.

```
@Comopnent
public class Main{
    @Reference
    ConfigurationAdmin ca;

    @Activate
    public void activate() {
        Configuration processConfig = admin.getConfiguration("
            autopilotsystem_impl.autopilot.autopilot", "?" );
        Configuration simulationConfig = admin.getConfiguration("
            autopilotsystem_impl.autopilot.simulation.simulation", "?" );
        Configuration rollCtrlConfig = admin.getConfiguration("
            autopilotsystem_impl.autopilot.rollcontrol.rollControl", "?" );
        ...
        Hashtable<String, ?> processProps = new Hashtable<>();
        processProps.put( "simulation.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.simulation.simulation)");
        processProps.put( "positionCtrl.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.rollcontrol.rollControl)");
        Hashtable<String, ?> simulationProps = new Hashtable<>();
        simulationProps.put( "throttles.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.simulation.throttles)");
        ...
        Hashtable<String, ?> positionCtrlProps = new Hashtable<>();
        rollCtrlProps.put( "command.target", "(de.unia.smds.uid=autopilot.
            autopilotsystem_impl.autopilot.rollcontrol.command)");
        ...
        rollCtrlConfig.update(rollCtrlProps);
        simulationConfig.update(simulationProps);
        processConfig.update(processProps);
    }
}
```

Listing 4.27: Configurational startup of AutopilotProcessImpl

Until now OSGi's configuration mechanisms were used to represent the initial state of a system, but provided no more configurability than the original `Main class` with its hard coded `main` method. Therefore, we will introduce AADL states in the next subsection and how those can also be represented with the aforementioned approach..

4.3.4. Modes in AADL

`Modes` in AADL can be compared to states, whereas **`mode`** transitions are equal to state transitions. Thus, **`modes`** in AADL resemble simple finite state machines as they are well known in the IT industry today. **`Modes`** can be used to describe either the state of hardware, e.g., **`processors`**, **`busses`** or other hardware elements defined by AADL, or can be used to describe different states of the software by using **`modes`** within AADL **`processes`** or **`threads`**. As we already restricted our approach to the software part of AADL, see section 2.2, we only examine **`modes`** in the context of **`processes`** and **`threads`**.

A **`mode`** in the context of a **`process`** or **`thread`** manifests itself as a configuration of contained components, **`connections`** and mode-specific properties and values. Such a configuration in the context of a **`process`** might be for example a specific subset of **`threads`** being active, or in context of a **`thread`** defining different execution modes or times, i.e., periodic versus aperiodic or a period of 200 versus one of 300 ms. Also **`connections`**, activated **`ports`** and several other properties of each of them can be defined to be included within a specific **`mode`** or not. In order to exemplify these mechanisms we depicted a **`modes`** declaration in AADL in Listing 4.28. We reused our previous example of an autopilot where we defined a **`process`** `autpilot` with two **`in event ports someEvent`** and **`someOtherEvent`**. We also defined its implementation **`autopilot.impl`** to contain several **`subcomponents`**, two of them being **`threads`** called **`simulation`** and `sensorinput`. In this example we added the declaration of two `modes`, i.e., `simulated` and `realworld`, describing the states the `process` can be in, being either within a simulated environment with artificial inputs or in a real-world scenario with real sensors delivering input to the autopilot. The initial or default `mode` is `simulated`, thus being the state the `process` would be in at startup. It remains in this mode as long as there is no event arriving at the `someEvent` port. Given the case such an event arrives, the `process` has to switch `modes` from `simulated` to `realworld`. Likewise it has to switch back to `simulated` if another event on the `someOtherEvent port` arrives. These `mode` transitions are defined in the lines 15 and 16 of our example.

Until now we defined two `modes` the `process` can be within and transitions between those `modes`, but did not explain the effect those `modes` have on the behavior of the `autopilot process`. This is done in lines 9 and 10 where we defined the `subcomponents simulation` and `sensorinput` to only be active in either the `simulated` or the `realworld mode` respectively. This means, in either state only one of those `threads` is active and forwards input accordingly to the rest of the system.

```
   process autopilot
 2     features
           someEvent: in event port;
 4         someOtherEvent: in event port;
   end autopilot;
 6
   process implementation autopilot.impl
 8   subcomponents
         simulation: thread simulation.impl in modes (simulated);
10       sensorinput: thread sensorinput.impl in modes (realworld);
         ...
12     modes
           simulated: initial mode;
14         realworld: mode;
           simulated -[someEvent]-> realworld;
16         realworld -[someOtherEvent]-> simulated;
   end autopilot.impl;
```

Listing 4.28: Declaration of modes in AADL

In our approach we decided to not implement all aspects and components of the AADL language, therefore we made several restrictions regarding the components and properties needed for `modes` and `mode` transitions to work properly.

First, in our example we placed the `mode` declaration in an implementation type declaration, whereas it also could be defined within a type declaration and by this passing on this `mode` declaration to all its inheriting component type declarations. We did this on purpose as we restrict the `mode` declarations in our approach to be defined within an implementation declaration as well as we explicitly excluded the possibility to reuse states of parent components via the `requires mode` subclause which is actually provided by AADL.

Second, we decided to include `event ports` for the special case of `modes` in our approach. This means, those can merely be used as triggers for `mode` switches, but can not be used to define arbitrary events that can be sent and received by components. This is done in order to enable a first impression of how `modes` can be mapped onto a generated framework, but omit the complications of a full incorporation of events, `event ports` and their semantics into our approach, as this would go beyond the scope of this work.

4.3.5. Mode Related Mapping

After introducing `modes` in AADL we will now explain how those `modes` and their transitions can be mapped onto OSGi components, properties and configurations.

`Modes` as they were presented in subsection 4.3.4 have to be mapped in two different ways onto an OSGi system. First, the composition of a system, where `subcomponents` or parts of components can be added or removed due to mode transitions must be adaptable. Second, OSGi components must be able to reflect changes of their properties due to `mode` transitions. We will first explain the composition of a system.

An AADL model usually describes the structural union of all possible `modes`, that is, for a given component all possible `subcomponents` and `features` known at design time. Additionally, all `modes` and `subcomponents/features` they en- or disable at runtime are also well-defined. Given the example depicted in Listing 4.28 the `process` implementation `autopilot.impl` defines two `subcomponents`, i.e., `simulation` and `sensorinput`, although only one will be active at a given point in time at runtime, i.e., `simulation` only during the `simulated mode` and `sensorinput` only during the `realworld mode`. Thus, AADL models always define a superset of `subcomponents` and `features` that are available at design time, whereas `modes` are a subset (or configuration) of all the available `subcomponents` or `features` available at runtime during a specific `mode`.

In order to map these semantics we use the configuration mechanisms of OSGi presented in subsection 4.3.2. Hereby, every single **`mode`** defined in an AADL model is translated into an according set of OSGi configurations. Thus, each component of the systems that is affected by the respective **`mode`** gets a configuration for this **`mode`**. This means, each possible **`subcomponent`** or **`feature`** which in code is represented as a reference to another OSGi component, is either made mandatory or optional by the configuration depending on whether or not is marked as active in the respective **`mode`**. By creating such a configuration for each component-**`mode`** combination for each OSGi component eventually all defined **`modes`** are represented by a set of these configurations.

To clarify this approach we created a sample `class Configurator` in Listing 4.29 which serves as a central entity, referenceable by any OSGi component that takes care of creating the right configurations for different **`modes`**. This `class` defines two different methods, i.e., `simulated` and `realworld` that are eponymous to the **`modes`** defined in Listing 4.28. Those two methods, if called by another component, take care of creating the right configuration(s) for the respective **`mode`**. Given the `autopilot` component received the event `someEvent` and wants to initiate a **`mode`** transition from `simulated` to `realworld`, then all it has to do is to call the method `simulated` in the `Configurator class`. This method first creates a configuration for the `autopilot` component. Then, it creates the according `target` and `cardinality.minimum` properties for the references of the `autopilot`. The `simulation` reference is set to a specific target and its cardinality is set to 1, thus making it a mandatory reference. `sensorinput` in turn is made optional by setting its `cardinality.minimum` to 0, thus enabling the `autopilot` component to start without having this dependency being resolved. Additionally, the `target` property of `sensorinput` is set to a predefined, never used constant `notarget` which ensures that it will not be resolved at any time. The other method, i.e., `realworld`, instead implements the exact opposite behavior, i.e., turning `sensorinput` into a mandatory reference and likewise making `simulation` optional. These mechanisms can be used for all relations wihtin an AADL model that are mapped onto OSGi references, i.e., **`subcomponents`**, **`features`** and **`connections`**.

```
@Comopnent(service = Configurator.class)
public class Configurator{
    @Reference
    ConfigurationAdmin ca;

    @Override
    public void simulated() {
        Configuration autopilotConfig = admin.getConfiguration("
            autopilotsystem_impl.autopilot.autopilot", "?" );
        Hashtable<String, ?> autopilotProps = new Hashtable<>();
        autopilotProps.put( "simulation.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.simulation.simulation)");
        autopilotProps.put( "simulation.caridnality.minimum", "1");
        autopilotProps.put( "sonsorinput.target", "(de.unia.smds.uid=notarget)
            ");
        autopilotProps.put( "sonsorinput.caridnality.minimum", "0");
        autopilotConfig.update(autopilotProps);
    }

    @Override
    public void realworld() {
        Configuration autopilotConfig = admin.getConfiguration("
            autopilotsystem_impl.autopilot.autopilot", "?" );
        Hashtable<String, ?> autopilotProps = new Hashtable<>();
        autopilotProps.put( "simulation.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.simulation.simulation)");
        autopilotProps.put( "simulation.caridnality.minimum", "0");
        autopilotProps.put( "sonsorinput.target", "(de.unia.smds.uid=
            autopilotsystem_impl.autopilot.simulation.sensorinput)");
        autopilotProps.put( "sonsorinput.caridnality.minimum", "1");
        autopilotConfig.update(autopilotProps);
    }
}
```

Listing 4.29: Mode transitions via configurations

Until now we merely used built-in mechnisms of OSGi to alter the runtime composition of a system, where we did not need any further generated or handwritten code in order to react to the different configurations deployed into the runtime. All changes are automatically handled by the SCR as only references, their cardinalities and optionalities are concerned.

However, there are also parts of an AADL system that can not be handled automatically by the SCR, e.g., properties. For our approach the most important properties that are used to alter semantic of AADL components are the **`Dispatch_Protocol`** and **`Period`** properties of **`threads`**. Those properties, which were explained in detail in subsection 2.2.4, alter the dispatch behavior of a **`thread`** and, in case of a periodic **`thread`**, its period. Those properties are implementation-specific and therefore can not be handled by predefined mechanisms of OSGi. Therefore, the generated code of our framework has to take care of this, where we reuse the configuration mechanisms of OSGi in order to tackle this challenge.

Each property that is used to alter the semantics of an AADL component has to be treated differently in its OSGi counterpart. Thus, we only exemplarily explain how the **`Period`** property is mapped onto code that reflects its semantic. In Listing 4.30 we depicted a **`thread`** similar to the one depicted in Listing 4.15 but extended it with an additional lifecycle method, i.e., `modified`, annotated with the OSGi `@Modified` annotation. This method, if defined, is called by the SCR whenever a new configuration is deployed into the system and is called with the respective configuration as a `Map<String, ?>` of properties. However the first time a configuration is made available for the respective component not the `modified` method is called by SCR but the `activate` method which is why this methods merely forwards every call to the `modified` method instead. Now, the `modified` method shows an excerpt from the original `activate` method of Listing 4.15. The only difference lies within the time that is given the respective `PeriodicTimer` to operate with. Whereas, in Listing 4.15 this time was fixed during the runtime and no changes could be made to a running system, we now declare this component to react to changes made by configurations deployed to the system at runtime. In this case, `ThreadImpl` first stops the currently running timer, then fetches the configured period from the given `props` and then creates a new `PeriodicTimer` which is subsequently started, now running with the period defined by the configuration.

```
@Component(
    configurationPolicy=REQUIRE,
    configurationPid="threadimpl"
    )
public class ThreadImpl{
  public AsyncEventHandler handler;
  public Timer timer;

    @Activate
    private void activate(Map<String, ?> props) {
        modified(props);
    }

    @Modified
    private void modified(Map<String, ?> props) {
      timer.stop();
      int period = (int) props.get("period");
      timer = new PeriodicTimer(null, new RelativeTime(period, 0), null);
      ...
        timer.start();
    }
    ...
}
```

Listing 4.30: Configurable Period property

This mechanism, i.e., reacting to configurations within a method annotated with `@Modified`, can be used to map all **`mode`**-specific property changes onto a system generated for OSGi.

4.4. Related Work

As to the best of our knowledge there is no work that can be compared directly with ours, meaning there is no work that targets AADL as source language or RTSJ as target language, let alone OSGi as a component framework on top of it. If viewed more abstractly then our work can be seen as model-driven development of service-oriented, SCSS, but there are only approaches partially tackling the challenges of this field. Therefore, we present different approaches for each combination of these three research fields in the following.

For the research area of model-driven development of service-oriented systems there exist plenty of approaches, mainly targeting the creation of service-oriented architectures and microservice architectures. Service-oriented architectures have experienced a hype around 2007 which is mirrored in the amount of papers published at that time on this topic. Microservice architectures currently experience the same hype as service-oriented architectures have 13 years before. Therefore, we picked approaches targeting each of those architectures to compare with our approach. [52] proposes a UML profile, called UML4SOA which they are using to generate code in multiple languages, e.g., Business Process Execution Language (BPEL), Web Service Description Language (WSDL), Java and Jolie. [52] has been part of a larger EU project called Sensoria [53] and *"[...] deals with the modelling of structural aspects of services, service orchestrations, policies, and other non-functional properties."* In contrast to our work, [52] targets service-oriented, distributed systems, communicating over a network and uses UML as source language. Our approach instead targets service-orientation within one system instead of a distributed system and also uses AADL as source language which is standardized in terms of semantic, visual and textual representation and syntax in contrast to the non-standardized semantic of UML and especially the proposed profile of [52]. Related approaches in the area of model-driven development of service-oriented systems are, according to [52], [54], [55] and [56], all of them also utilizing UML as source language and targeting distributed systems.

The research area of microservice architectures is currently flooded with approaches regarding generation of microservice configurations regarding specifc metrics like elasticity [57], the generation of several microservices to form a complete backend system [58] or even complete microservice architectures with API gateways and service discoveries included [59]. Common to all of them is the generation of a service-oriented, distributed system in contrast to our approach of a service-oriented system within one process. Additionally, none of them uses a standardized source language that could easily be used for verification or validation within a certification process. [58] uses its own domain language called JHipster Domain Language (JDL) for the description of all applications, deployments, entities and their relationships within a single file. [59] facilitates Maven [60] and its archetype system for the generation which is a purely command line based and thus a non-standardized approach.

The research area of model-driven development of SCSS usually encompasses two subareas, i.e., approaches that analyze models of SCSS in order to verify or validate them [61, 62, 63] and approaches that generate code, but usually only test code. The first area only partially overlaps with our approach as we rely on AADL for those disciplines, but do not directly target issues like model-driven verification or validation. The second subarea is directly addressed by our approach in terms of generating a working code base that can be run on any real-time JVM, whereas the majority of research approaches merely targets the generation of tests for such a system, e.g., [64] or [65].

Approaches directly targeting the generation of a SCSS itself are rather sparsely represented. One of these is for example [66] which uses a specialized subset of UML, e.g., component diagrams and real-time state charts, in order to describe systems which is again a non-standardized source language in contrast to AADL. They especially target the reduction of state explosion by a compositional reasoning approach which is tackled by our approach by use of contracts, see chapter 5. Another very well-known approach is the usage of Matlab Simulink [48] which also does not support a high-level language like Java as target language. Besides, [48] encourages modeling of business logic, which is explicitly excluded from our approach as we regarded this as too low-level to enhance productivity.

Service-oriented systems are in general rather hard to find in the domain of SCSS, with one exception being the automotive domain. There, some approaches emerged that tackled challenges of this domain by utilizing concepts of service-oriented architecture, e.g., [67] or [68]. But again, those approaches merely examined service-orientation within a distributed system, not within one system as our approach does.

4.5. Evaluation

For the evaluation of our approach we again use the running example presented in section 2.1. In contrast to our former approach we generated the code this time as an OSGi system. The golden standard against which this generated system is compared is still the handwritten implementation of section 3.4. As we are still only interested in the parts of the code that are related to either structure, timing or communication, we stripped the handwritten as well as the generated code of all logic related parts. We then applied the same quantitative tests as were already applied in subsection 3.4.1 and additionally added a test that detects the performance impact of OSGi related code onto the generated system of our current approach. We then examined the impacts of using OSGi for a system with `modes` and how much time the `ConfigurationAdmin` and the overall OSGi runtime takes to de- and reconstruct a simple system. Finally, we conducted a qualitative analysis of different typical scenarios once solved with plain RTSJ and once with an OSGi solution.

4.5.1. Quantitative Evaluation

First, we compared the SLOC of the different approaches. As can be seen in Figure 4.3 the amount of manually written code stays the same as for our first approach and therefore roughly half the size of a purely handwritten solution. As in subsection 3.4.1 this implies a 50% reduction in time of writing, which is a tremendous benefit if applied to larger systems. We were also able to reduce the amount of generated code in our current approach compared to the former one, resulting in an approximately 9% smaller, generated code base.

In Figure 4.4 we compared the messages sent per second of the different approaches. The mechanisms that are used to transmit messages from one component of the system to another differ between the three approaches. The golden standard solution usually results in calling a method directly on the target component that delivers the message to this component which is obviously the fastest, but also the least adaptive approach. As can be seen in Figure 4.4 this approach leads to roughly 315 million messages sent per second which is more than three times the number of messages sent by the aadl2rtsj approach. The aadl2rtsj approach facilitates the concept of a CB as explained in subsubsection 3.2.3.3. This CB internally has usually a large generated switch-case statement that routes the message to its respective receiver component. This ap-

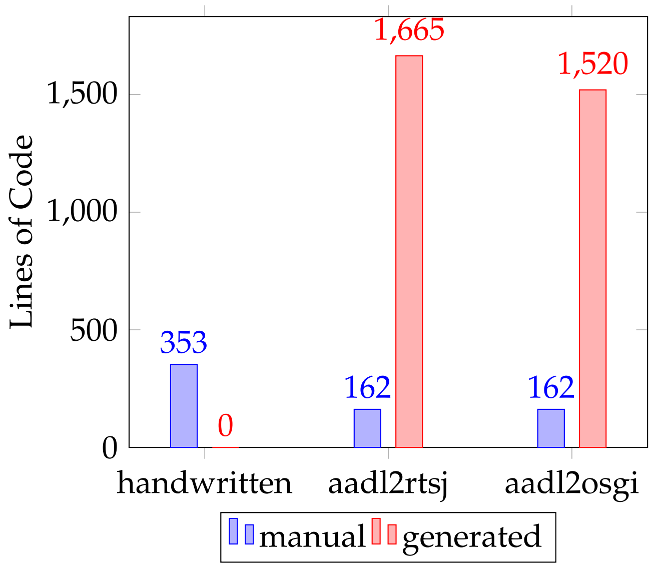

Figure 4.3.: SLOC comparison between handwritten, aadl2rtsj and aadl2osgi solution

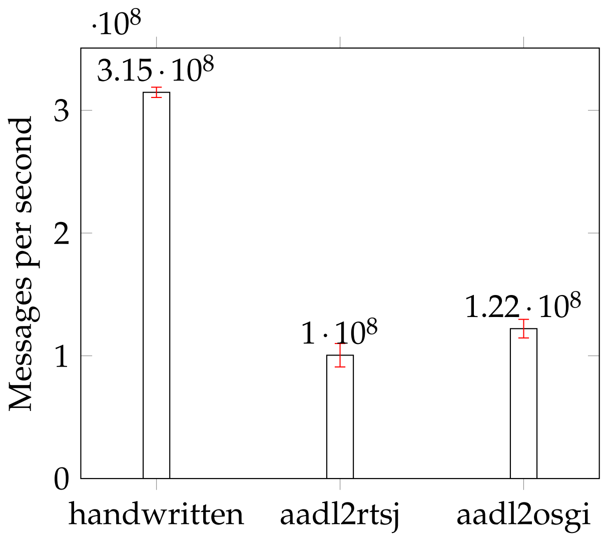

Figure 4.4.: Messages per second comparison between handwritten, aadl2rtsj and aadl2osgi

proach degrades in speed depending on the position of the receiver within the switch-case statement as well as of the number of `connections` between sender and receiver, each resulting in another switch-case statement, i.e., a higher calldepth.

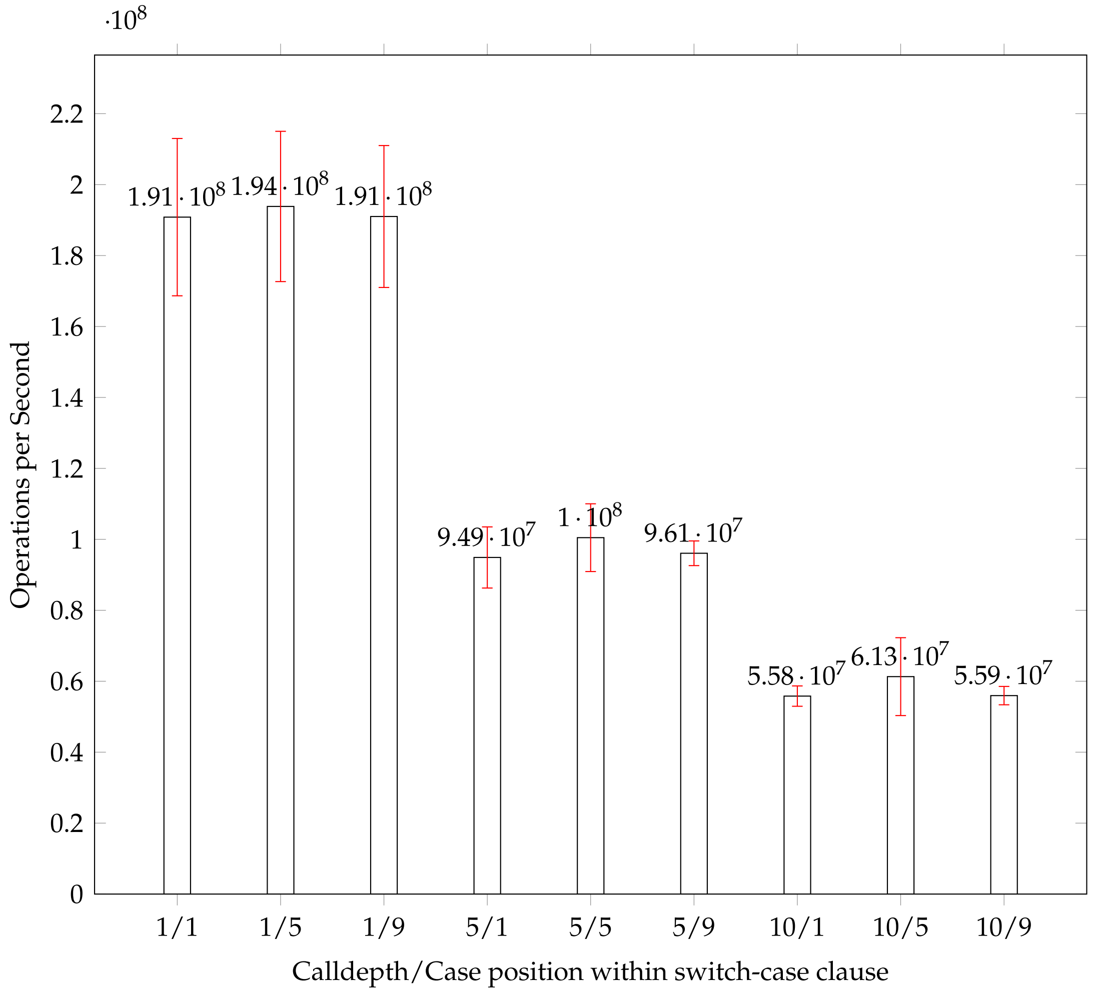

Figure 4.5.: Message throughput of aadl2rtsj solution with differnt calldepth/-case position combinations

The number presented in Figure 4.4 therefore is only one, i.e. the one for calldepth 5, of a number of different possibilities, all depicted in Figure 4.5. Here, we can see that a position in the middle of the switch-case statement is beneficial in terms of execution time or throughput. The calldepth of course has a direct impact on throughput, meaning the higher the calldepth the longer it takes to transmit a message and therefore less throughput can be generated.

Our current solution enhances the former approach as we got rid of the switch-case statements for each step in a call chain. By leveraging OSGi's mechanisms of service composition a message transmission at runtime results in a chain of

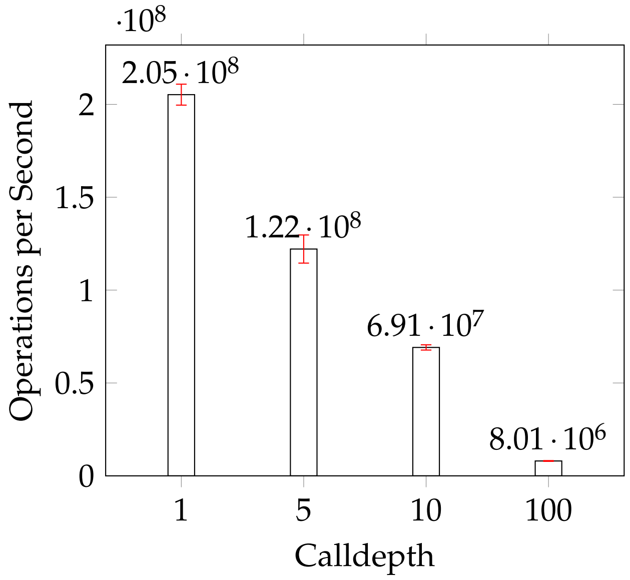

Figure 4.6.: Throughput aadl2osgi with different calldepths

method calls, which removes the overhead of the former switch-case statement. Now, the execution time of a message transmission merely depends on the number of chained methods which is also reflected in Figure 4.4, where our current approach is approximately 21% faster than the former aadl2rtsj approach. In Figure 4.6 we can see the direct impact of chain length on throughput and therefore transmission time for messages. In Figure 4.4 we always have chosen a calldepth of 5 for both, aad2rtsj and aadl2osgi, in order to be able to directly compare the approaches, but as mentioned before each case can be different.

In contrast to our former approach, aadl2osgi added OSGi as an additional layer of abstraction on top of the JVM. While providing better means of modularization and decoupling it also has a significant impact on overall performance and memory consumption. Many objects are not any longer created directly via the `new` keyword, but by mechanisms explained in detail in subsection 4.3.5. This enables us to represent states of the system as configurations, but intitialization of such a state can be much slower than a handwritten solution. In order to measure the impact of OSGi on object (service) creation we implemented a benchmark measuring the throughput of OSGi in terms of service creations per second. Each service in OSGi is created with a configuration, namely a `Map` with several key-value pairs. These configurations can differ in size depending on the configurability of the respective service. In Figure 4.7 we depicted the

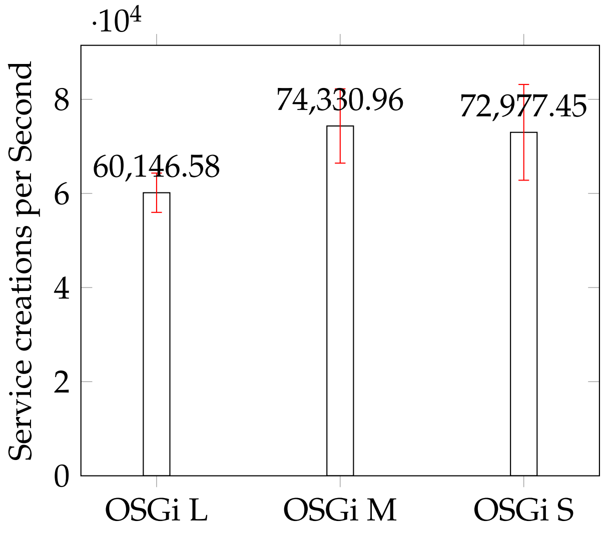

Figure 4.7.: OSGi's throughput for creating services with different sizes of configurations

results for service creations with different sizes of configuration objects, i.e., a large configuration containing 20 key-value pairs (OSGi L), a medium configuration containing 5 key-value pairs (OSGi M) and a small configuration containing a `null` configuration (OSGi S). The size of a configuration has a direct impact on the time needed to create a service. The larger the configuration, the slower the service creation, resulting in 60.146 service creations per second for large configurations and 74.330 service creations per second for medium configurations. `null` configurations seem to be handled differently from non-`null` configurations, as in contrast to the expected highest throughput this type of configuration took more time than the medium one. Probably, the OSGi framework incorporates additional `null` checks and the creation of an empty `Map` if the given configuration for a service is `null` which would explain the lower throughput.

Apart from time consumption another important key figure is the amount of memory consumed by each approach. Therefore, each benchmark was executed with a profiler attached so that memory allocation and deallocation could be tracked. For the handwritten solution this resulted in 16 kilobytes allocated for each run which lead to allocation peaks of over 1 gigabyte per second during testruns. Although each single testrun for the OSGi solution did allocate far more memory, ranging from 4,5 kilobyte for small configurations to 6,5 kilobyte for large configurations the allocation peaks only reached 313 to 363 megabytes

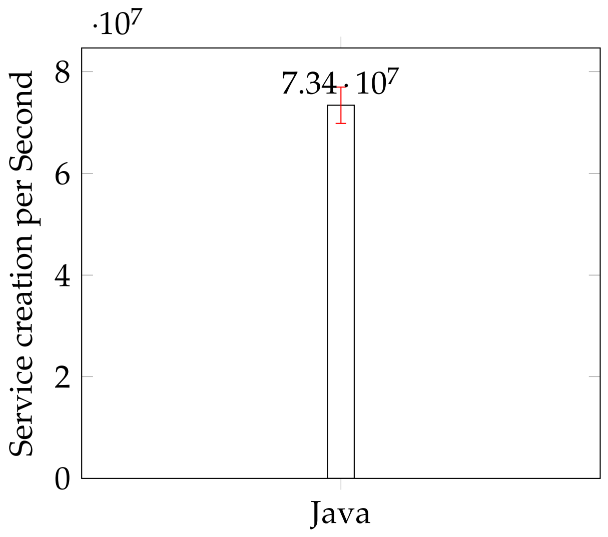

Figure 4.8.: Java throughput for creating objects

per second respectively.

In summary, it can be said that the impact of OSGi on object/service creation is massive in comparison with the creation of a simple object, as depicted in Figure 4.8. Although, it also has to be said in all fairness, that this comparison is rather coarse, as the creation of a service incorporates the creation of additional metadata, several checks for configuration parameters on how (and if) the service should be created and several more steps, where in contrast a object creation in java is a primitive operation. The numbers presented in Figure 4.7 therefore should rather be taken into consideration when analyzing the impact of OSGi on state switches of a system than to be used in a direct comparison between a handwritten and a generated solution.

4.5.2. Qualitative Evaluation

Aside from a purely quantitative evaluation we also played through different scenarios that put the generated, modular solution in contrast to a handwritten one in terms of maintainability and runtime reconfigurability. Also a comparison with our former approach is executed. For both we will reuse the running example defined in section 2.1.

Maintainability through Modularity The first use-case under consideration is the addition of a component and will show the increased maintainability of the generated modular code base. In subsection 3.4.2 we compared the addition of a component in a pure handwritten solution to the addition of a component written in AADL and the subsequent generation of it. For this we chose the example of a logger that is added to the system after it has been initially generated or written. We showed that the addition of this logger and its **`connection`** to the rest of the existing system would be rather error prone in a purely handwritten solution. Many existing `classes` would have to be altered as well as additions would have to be made in several different locations that might not be obvious to a developer. Therefore, we claimed our aadl2rtsj approach being superior in such cases, as the addition of the component in AADL is a rather small task, is well supported by standardized IDEs and the developer gets visual support through a standardized graphical representation that for example makes missing **`connections`** more obvious.

However, in order to add the logger component not only the component itself has to be generated, but also other components it is linked to via **`connections`** due to the implementation details of the generated solution, i.e., how immediate **`connections`** are handled within a CB. In Listing 4.31 we depicted a simplified version of the generated code for the **`AutopilotConnectionBroker`** which takes care of transmitting data from the **`mixThrottlesControl`** to the newly added logger component. In our fictional case this CB already has been generated a first time when the original system without our newly added logger component had been generated. Therefore, a version of this CB already existed in the code base which now has to be altered in order to contain code that takes care of transmitting data from **`mixThrottlesControl`** to the newly created logger component as well as code that takes care of timing as we defined the new **`connection`** between both components to be **`immediate`**. This is depicted in Listing 4.31 by the additional `case "con_1"` as well as the added synchronization object `syncObjectCon1` and the additional `Logger` member variable.

```java
public class AutopilotConnectionBroker {
    private Object syncObjectCon1 = ...;
    private Logger logger = ...;

    public void sendOnConnection(String connection, Object data) {
    switch (connection) {
    case "con_1":
        logger.inThrottleCmd((ThrottleCmd) data);
        syncObjectCon1.notify();
        break;
      ...
    }
```

```
  }
```

Listing 4.31: Simplified immediate connection from mixThrottlesControl to logger component

Thus, the purely generated solution not only does not cover edge cases, but also has to be regenerated in order to add a newly defined component. This results mainly from immediate **connections** which, in Java, are represented using synchronization objects on which both, sender and receiver, can synchronize to transmit data within one dispatch cycle, see section 2.2.4.3. If there is an additional immediate **connection** added, then not only has this **connection** to be represented by an additional switch-case statement in the CB, but also has the sender to be extended with an additional reference to the mandatory synchronization object. Thus, a preexisting `class` has to be altered and the whole system must be regenerated again. Therefore, our former approach indeed is better than a handwritten solution with regard to error susceptibility but lacks some important properties to be maintainable, i.e., strict encapsulation and a clear separation of concerns.

Our new approach renders such regeneration obsolete as it generates a truly modular system in which `classes`/services and references between them can be altered afterwards within specific boundaries which improves the overall maintainability of the system. For example, in aadl2osgi we are able to add an additional **connection** to an existing **port** easily by creating a new service representing this **connection** and changing the configuration of its source **port** by deploying an appropriate configuration to the running system.

```
@Component(
    configurationPolicy=REQUIRE,
    configurationPid="handwritten.Con1",
    property={"de.smds.unia.uid=handwritten.Con1"}
    )
public class Con1 implements PortConnection<Integer>{
    @Reference
    IFWOutPort<Integer> source;

    ...
}

@Component
public class MyConfigurator {
    ...
    @Activate
    private void activate(){
        Configuration configuration = admin.getConfiguration("handwritten.Con1
            ", "?" );
        Hashtable<String, ?> ht = new Hashtable<>();
        ht.put( "source.target", "(de.unia.smds.uid=mixthrottlescontrol.
            throttlecmdout)");
        configuration.update(ht);

        Configuration configuration2 = admin.getConfiguration("de.unia.smds.
            uid=mixthrottlescontrol.throttlecmdout", "?" );
        Hashtable<String, ?> ht = new Hashtable<>();
        ht.put( "outgoing.target", "(de.unia.smds.uid=handwritten.Con1)");
        ...
        configuration2.update(ht);
    }
}
```

Listing 4.32: Adding a handwritten connection to a preexisting port

In Listing 4.32 we depicted such a solution for a handwritten case. First, we created a new **`connection`** `Con1` reusing the `interfaces` provided by aadl2osgi, i.e., `PortConnection` for the **`connection`** itself and `IFWOutPort` for its source port. This way we ensure, that the generated code can interact with our handwritten one. In the next step we first configure `Con1` to target **`mixThrottlesControl.-throttlecmdout`** as source **`port`** and then we configured `ThrottleCmdOut` to target `Con1` as an outgoing **`connection`**. Of course this is a simplified configuration as we usually would have to ensure that `ThrottleCmdOut` would also target all other **`connections`** it targeted before we added `Con1`, but this can be easily done by copying the configuration and then adding the additional target `Con1`.

Aside from now being able to extend a generated system without regenerating the already existing code, the aforementioned benefits of section 3.5 still hold true. AADL remains superior in the majority of cases regarding SCSS design regarding introduction of errors, whereas plain Java remains superior in edge cases maybe not covered by AADL components and semantics.

The interchangeability of existing components as well as the addition of new ones mostly stems from the heavy use of `interfaces` within OSGi, thus guaranteeing strong encapsulation and separation of concerns within a code base. This strong encapsulation again is a major advantage of aadl2osgi over aadl2rtsj as now a recertification of a system is by far easier than with our former approach. Existing components within a generated system are not altered anymore when new functionality is added and by leaving the old code untouched recertification is not a topic anymore. Only the parts that are added have to be certified to work correctly, also in interplay with existing components, thus freeing developers of a SCSS from recertifying old parts of the system.

Runtime Code Replacement Additionally, by using OSGi as the underlying framework a system is now able to support **hot updates**, i.e., updates of the code at runtime. This feature is possible due to the strictly defined lifecycle rules of bundles, see subsection 2.4.2, services, see subsection 2.4.3, and the dynamism that inherently is added to all services via lifecycle methods and dynamic adding and removing references to other services. This way each service is well aware of its changing surroundings in terms of other services and bundles that may come and go during runtime. Each service and bundle therefore is written with this dynamism in mind and is able to cope with changing references or being gracefully stopped by the framework at all.

In order to examine this feature more in depth we again use the running example of section 2.1. In this example we alreday generated a running system from a sufficiently detailed AADL model. Each controller of the autopilot is deployed to the OSGi framework as a separate bundle, i.e., .jar file. We assume there is a bundle for `AltitudeController` with a symbolic name "altitudecontroller.fixed", which controls the altitude of our quadrocopter by keeping it always at the fixed height above ground of one meter.

Now, imagine the requirements changed for our quadrocopter and we need to change the behavior of our `AltitudeController`, i.e., it should not keep a fixed height above ground, but react to commands from the outside which tell it the height to keep. In our previous approach and also in a handwritten solution we would have to model/write the software, compile it, package it, somehow deploy it to our quadrocopter and then launch the quadrocopter with the new software. For this to work, we usually have to shut down the quadrocopter for a while, i.e., the time it takes to deinstall the old autopilot and install the new autopilot. As this is a rather cumbersome procedure and demands physical access to the quadrocopter. However, we would prefer to be able to do this procedure without having the quadrocopter to be landed, shut down and restarted subsequently. Although not absolutely necessary for a quadrocopter, other embedded systems might benefit far more from such a functionality, especially in cases where access to such devices is hard to achieve.

In contrast, with aadl2osgi we are now able to do just that. Not only are we able to replace the software without physical access to the device, but we could do so even in midair. The code base generated by our approach can be exchanged during runtime by the use of an additional service standardized by the OSGi alliance in their compendium specification [44], i.e., the `Configurator` [69]. Once installed in the framework this service takes care of deploying configurations of a bundle into the runtime via using the `ConfigurationAdmin`. The additional value of `Configurator` is its standardized behavior which enables us to not only deploy a bundle into the system, but also a corresponding configuration for this bundle which, if placed in the right location within the bundle's folder structure, is found by the `Configurator` and deployed to the runtime accordingly. From there on the mechanisms that take care of registering and adding new services within the runtime are the same as already described in subsection 4.3.5.

```
@Component(
    configurationPolicy=REQUIRE,
    configurationPid="autopilot",
    property={"de.smds.unia.uid=altitudecontrol"}
    )
public class Autopilot {
    @Reference
    AltitudeController altitude;
    ...
}

@Component(
    configurationPolicy=REQUIRE,
    configurationPid="altitude.dynamic",
    property={"de.smds.unia.uid=altitude.dynamic"}
    )
public class AltitudeControllerDynamic implements AltitudeController {
    ...
}
```

Listing 4.33: Simplified Autopilot and its reference to MixThrottlesControl

Now all a developer has to do is generating the code for the new `AltitudeController`, implement the business logic and write the configuration for the existing `Autopilot` to use the new `AltitudeController` instead of the old one. In Listing 4.33 we depicted a simplified version of an `Autopilot` which holds a reference to an `AltitudeController` as well as the newly created, dynamic `AltitudeControllerDynamic` implementation. Given such an implementation, the configuration for `Autopilot` would merely contain the key-value-pair `altitude.target=(de.smds.unia.uid=altitude.dynamic)` which advises the `Autopilot` to use the dynamic implementation instead of the static one. This configuration is contained within the .jar file that is subsequently deployed to a running system. After the framework installed this .jar the `Configurator` scans the bundle for configurations and deploys those via the `ConfigurationAdmin` into the running system. The runtime then picks up this configuration, matches it to an instance of `Autopilot` and forces the implementation to change its reference of `AltitudeController` to the dynamic one as soon as the static one is stopped during the deployment process.

4.6. Conclusion

In this section we presented an adoption of our former approach aadl2rtsj, targeting not only a high-level language as target language for our code generation, but also a modular code base by design. This approach enables developers of a SCSS to not only shift structure, timing and communication-related concerns into design phase, but also enables them to create a highly modular, runtime reconfigurable and, most important, a more easy to maintain system by design. Advantages explained in our former approach still hold true, i.e., being able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices, while new advantages, i.e., enhanced maintainability and better configurability are added on top.

Within the evaluation of this approach we have shown that those additional benefits come with a cost, e.g., configurability comes with an additional computing cost as well as an increased memory consumption, but at the same time other aspects of the generated code have been improved by the new architecture of the generated code, e.g., throughput of messages has been increased by over 21%, meanwhile shrinking the size of the generated code by 9%. Applied to the autopilot implementation of our former approach, this means better performance during one state and the possibility to switch between several states, e.g., flying and landing, but with an added cost in terms of computing time and memory during these transitions.

Another aspect targeted by enhanced reconfiguration and extendability / maintainability is the certification process of SCSS. While the former approach required a developer to regenerate the whole code base for even the smallest change, it is now possible to enhance an existing system by adding/altering only specific parts of the overall system, due to its modularity.

This aspect will be even further enhanced in the following chapter by enabling a developer to show that newly added components are semantically equivalent to the components they alter or replace.

5 Certifiability through Contracts

5.1. Motivation

The last objective left of those defined in chapter 1 is "Certifiability". As SCSS are usually subject to certification processes and certification of a system usually is a cost- and time-intensive process, developers mostly want to certificate only updates or parts that are newly introduced to an existing system instead of re-certificate the whole system. Although the former chapter already showed how to cleanly separate code and therefore, makes it easier to show none of the other code parts are afflicted if new code is added, it does not provide means by which a developer can show that the replacement for an existing code fragment does the same as the replaced one. In order to enable such a partial certification one has to show that the newly added or updated parts are semantically the same as the ones before and that there are no unwanted impacts on the overall system. Therefore, the update or new part of a system should be (semi-)automatically demonstrable to be semantically equivalent to the replaced/updated part. Usually, this is easy as long the system under development is small, but becomes harder the larger the system becomes. As already shown in chapter 1 SCSS or embedded systems in general tend to become larger and larger, thus becoming more complex and harder to certify. A developer of a SCSS often is confronted with components whose interrelationships and behavior he can hardly intuitively grasp anymore, i.e., systems with a too large number of possible states they can get into.

In order to gain access to the properties of such a system a developer usually decomposes such a system into smaller components, as the behavior and interaction of those encapsulated components is relatively easy to understand and describe in contrast to the overall system. With our aforementioned approaches we succeeded in designing systems in a way that are by design modular. Every model element is translated into a self-contained component, thus making each component easier to understand and more maintainable as the system as a whole. By utilizing RTSJ, i.e., Java, and OSGi we encapsulated each component with a syntactically well-defined `interface` that formally describes the types that methods are expecting as parameters as well as types that are returned to a caller of those methods.

This syntactic definition of an `interface` fulfills the requirements for Level 1 contracts as defined by [70]. As shown in Figure 5.1, [70] defined *"four levels of increasingly negotiable properties"* of contracts between components which start with non-negotiable properties on a syntactic level, i.e., interface definition languages or programming languages as Java, and progresses to dynamically negotiable quality-of-service level properties, e.g., response time of specific methods.

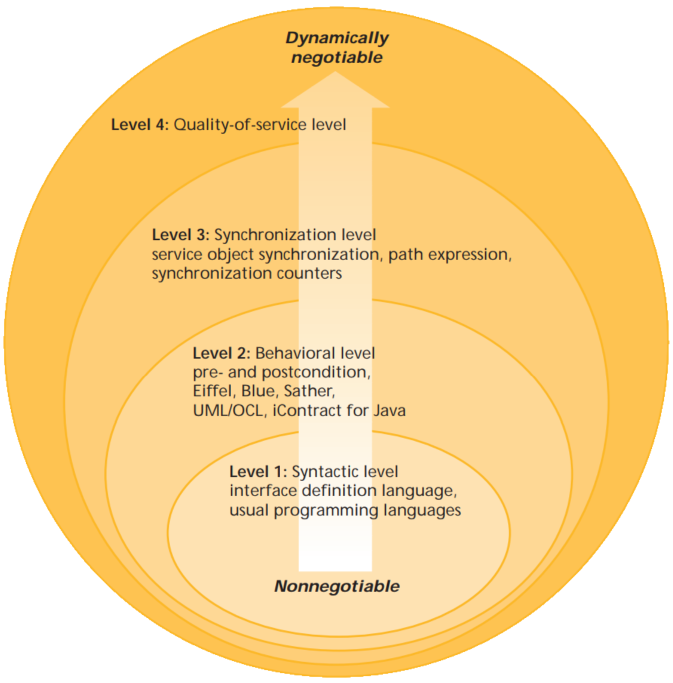
Dynamically negotiable
Level 4: Quality-of-service level
Level 3: Synchronization level
service object synchronization, path expression, synchronization counters
Level 2: Behavioral level
pre- and postcondition, Eiffel, Blue, Sather, UML/OCL, iContract for Java
Level 1: Syntactic level
interface definition language, usual programming languages
Nonnegotiable

Figure 5.1.: The four contract levels [70]

Between syntactical and quality-of-service properties there are also behavioral and synchronizational properties. Those could be pre- and post-conditions for behavior or automatons for synchronization.

Those different types of contracts can be used to even enhance the description of a component as we did define them in the previous chapters. This would enhance the possibility for a developer of a SCSS to test two different components for their semantical equivalence, thus easen the demonstration of semantic equivalence needed for partial certification of such a component. Interconnections between different components could be (semi-) formally described by these contracts and thereby reduce the work of showing their semantic equivalence to demonstrate the fulfilling of those contracts.

As we already provide components by design through the usage of OSGi and provide contracts on a syntactical level by using Java `interfaces` as interfaces between components, we only have three other levels of contracts left in order to define contracts between components, i.e., behavior, synchronization and quality-of-service. However, in this section we will provide contracts only targeting two of these three level, i.e., behavior and quality-of-service, due to the extend of this work and restrictions of our chosen language, i.e., RTSJ, and our chosen component framework, i.e., OSGi.

The rest of this section is structured as follows: section 5.2 provides a definition of contracts as they are used in the following approach as well as an detailed explanation of the mechanisms of OSGi used to implement contracts for our existing approach. section 5.3 details our mapping of different contract types onto sets of attributes of a capabilities in OSGi and shows how to enforce those contracts at runtime. In section 5.4 we compare our approach to existing approaches in the same field of research. Finally, we evaluate our approach by applying it to the previously used running example of an autopilot and showing the effects of contracts on replacing, updating and adding components.

5.2. Contracts and additional OSGi Basics

In the following subsections we first explain the term "contract" within the context of this approach. This is followed by a detailed explanation of OSGi's requirements and capabilities as our approach heavily depends on the inner workings of OSGi's dependency resolution mechanisms. Additionally, we introduce OSGi's service hooks and Java's proxying capabilities which will be used for runtime enforcement of our contracts.

5.2.1. Contracts

Contracts in the context of software engineering have been first made popular by Bertrand Meyer in [71] and were based on pre- and post-conditions as well as invariants for functions. This notion of contracts has been implemented by some programming languages natively, e.g., D, Eiffel or even Ada in its 2012 version. However, contracts in general can encompass more than the aforementioned pre-/post-conditions. [72] provides an extensive overview of the current state of contracts within research as well as in industrial contexts. Aside from providing their own meta-theory of contracts, they identify two major research areas regarding contracts, i.e., assume/guarantee contracts and interface theories.

Assume/Guarantee contracts encompass all sorts of contracts that on the one hand make assumptions that characterize a valid environment of a component and on the other hand offer guarantees for the respective components as long as they are used in such an environment. [72] further divides assume/guarantee contracts into data flow assume/guarantee contracts and synchronous assume/guarantee contracts. As we prefer a rather hands-on approach on contracts we only use pre- and post-conditions as assume/guarantee contracts in our context.

Interface theories, as described in [72], usually treat components as input/output automatons. As such their behavior can be defined as interface state automatons. Interface state automatons describe all possible states and transitions between them that an implementation of a specific interface can get into. Transitions usually are triggered by outside events, e.g., a method call, but also inner events are possible, e.g., a timer that triggers a state transition after a specified amount of time. Interface state machines offer developers of SCSS means to reduce the number of possible states a component can get into by hiding possible in-between states of an implementation behind a facade, i.e., an interface. This reduction of possible states later on easens testing, as only the official states of a component as described by its interface state machine have to be tested. We exclude interface state machines from our approach, due to restrictions of the requirement and capability mechanism that is used for our approach.

5.2.2. Requirements and Capabilities

According to the official OSGi core specification [73] *"OSGi dependency handling is based on a very general model that describes the dependency relationships. This model consists of a small number of primitive concepts:*

- ***Environment*** *- A container or framework that installs Resources.*
- ***Resource*** *- An abstraction for an artifact that needs to become installed in some way to provide its intended function. A Bundle is modeled by a Resource but for example a display or secure USB key store can also be Resources.*
- ***Namespace*** *- Defines what it means for the Environment when a requirement and capability match in a given Namespace.*
- ***Capability*** *- Describing a feature or function of the Resource when installed in the Environment. A capability has attributes and directives.*
- ***Requirement*** *- An assertion on the availability of a capability in the Environment. A requirement has attributes and directives. The filter directive contains the filter to assert the attributes of the capability in the same Namespace."*

Dependencies described by this model can vary widely, i.e., `Import-Package` and `Export-Package` relationships between bundles or dependencies between services and components or even OSGi contracts as explained later in section 5.4. Each of those types has an according namespace in OSGi. Namespaces that are used for predefined types of dependencies are listed in the Framework Namespaces Specification [74]. This specification defines the semantics of these predefined namespaces.

Namespaces are denoted by a name, usually in a form similar to a unique qualifying `class` name, i.e., `de.unia.smds.namespace`, and are in general used to group a set of attributes that in turn are used to describe a capability. Attributes can have a type, e.g., primitive types like `int`, `float` or `boolean` or `lists` of primitive types or a `Version`. A `Version` is a `String` in the form of `major.minor.macro`, where `major`, `minor` and `macro` are positive Integers. `Version` is treated as a semantic version [75] which implies it has a natural ordering, e.g., 1.2.0 is smaller than 1.3.0. If no type is given, the type is assumed to be `String`.

A capability defined this way then can either be provided or required by any given resource. A resource providing a capability defines a `Provide-Capability` header in its manifest followed by the namespace of the capability it provides, followed by at least all mandatory attributes with values attached to each of them. On the other side, if a resource requires a given capability, then usually it defines a `Require-Capability` header in its manifest, followed by the namespace of the respective capability, followed by at least one filter for possible values of an attribute of the given capability. Requirements also can be declared to be usable by more than one other resource through their `cardinality` attribute. Additionally, requirements can be declared to be mandatory or optional.

The aforementioned filters in OSGi are based on LDAP filters [36]. The string representation of an LDAP search filter uses a prefix format and is defined by the grammar defined in Listing 5.1.

```
filter      ::= '(' filter-comp ')'
filter-comp ::= and | or | not | operation
and         ::= '&' filter-list
or          ::= '|' filter-list
not         ::= '!' filter
filter-list ::= filter | filter filter-list
operation   ::= simple | present | substring
simple      ::= attr filter-type value
filter-type ::= equal | approx | greater-eq | less-eq
equal       ::= '='
approx      ::= '~='
greater-eq  ::= '>='
less-eq     ::= '<='
present     ::= attr '=*'
substring   ::= attr '=' initial any final
initial     ::= () | value
any         ::= '*' star-value
star-value  ::= () | value '*' star-value
final       ::= () | value
value       ::= <see text>
attr        ::= <see text>
```

Listing 5.1: Grammar for LDAP search filters [76]

This grammar enables us to create simple filters, e.g., a check for an exact match of an attribute value or its presence or absence, respectively. However, also more complicated filters are possible, e.g., a check if the value of an attribute is within a specified range or in general a combination of several simple filters. Even approximations and wildcards are supported which can be a powerful instrument in order to create highly complex filter conditions.

The interplay of requirements, capabilities and filters is exemplarily shown in Listing 5.2 and Listing 5.3. In this example we decided to use computers and games to depict the interplay between provided capabilities and filters for requirements. Therefore, we defined two attributes:

- *graphiccards:* **a list of strings** naming the graphic card provided by the computer
- *ram:* **a positive integer** specifying the amount of RAM build into the system

Both attributes are declared within the namespace `de.unia.smds` and the capability itself has the name `system`. In Listing 5.2 we defined three systems, i.e., a `High-End-System`, a `Medium-System` and a `Low-End-System`. The `High-End-System` offers two graphic cards, i.e., one from NVidia and an onboard one, and 32GB RAM, the `Medium-System` offers a Radeon and an onboard graphic card as well as 16GB RAM and finally the `Low-End-System` offers merely 4 GB of RAM and an onboard graphic card.

```
Bundle-Symbolic-Name: High-End-System
Provide-Capability:
  de.unia.smds.system;
    de.unia.smds.system=system;
        graphiccards:List<String>="nvidia,onboard";
        ram:Long=32

Bundle-Symbolic-Name: Medium-System
Provide-Capability:
  de.unia.smds;
    de.unia.smds.system=system;
        graphiccards:List<String>="radeon,onboard";
        ram:Long=16

Bundle-Symbolic-Name: Low-End-System
Provide-Capability:
  de.unia.smds;
    de.unia.smds.system=system;
        graphiccards:List<String>="onboard";
        ram:Long=4
```

Listing 5.2: Definition of hardware capabilities of a system

We now define two games that require specific hardware in order to be playable, i.e., a `High-End-Game` and a `Casual-Game`. The `High-End-Game` requires at least 16GB RAM or more and either a NVidia or a Radeon graphic card in order to be playable. This is expressed in Listing 5.3 as a filter in form of `(&(|(graphiccard=nvidia)(graphiccard=radeon))(ram>=16))`, combining three simple filters into one. When this filter is applied, only the `High-End-System` and `Medium-System` of Listing 5.2 would satisfy the requirements posed by `High-End-Game`.

In contrast, the `Casual-Game` merely requires at least 4GB RAM but no specific graphic card. Therefore, the filter for the `Casual-Game` makes use of a wildcard operator ∗ which is equivalent to an existential quantifier, whereby every system defined in Listing 5.2 would satisfy this requirement.

```
Bundle-Symbolic-Name: High-End-Game
Require-Capability:
  de.unia.smds.system;
    filter:="(&(de.unia.smds=system)(|(graphiccard=nvidia)(graphiccard=radeon)
        )(ram>=16))"

Bundle-Symbolic-Name: Casual-Game
Require-Capability:
  de.unia.smds;
    filter:="(&(de.unia.smds=system)(graphiccard=*)(ram>=4))"
```

Listing 5.3: Definition of hardware requirements of a game

As the capability definitions shown in Listing 5.2 have to be written by hand and the IDE does not offer any support in this regard, we leverage a mechanism newly introduced in OSGi's R7 specifications to enhance the handling of those capabilites: package annotations.

Package annotations have been introduce in OSGi's R7 specification in order to be able to declare capabilities directly in the code, thus getting partial support by the IDE, instead of writing them entirely by hand which is rather error-prone. Therefore, the OSGi specification defined several so called meta annotations, i.e., annotations that can be used to annotate other annotations that can then be applied to `classes` or packages and makes the corresponding bundle automatically provide the capability defined in the meta annotation. Those meta annotations are:

- **@Capability** - Define a capability for a bundle

- **@Attribute** - Mark an annotation element as an attribute

With those meta annotations the capabilities presented in Listing 5.2 can now be written as shown for `High-End-System` in Listing 5.4.

```
@Capability(namespace = " de.unia.smds.system")
public @interface  SystemCap {
    @Attribute("graphiccard")
    String[] graphiccard();

    @Attribute("ram")
    long ram();
}

@TyperangeCap(graphiccard={"nvidia","onboard"}, ram=32)
public HighEndSystem {}
```

Listing 5.4: Definition of system capabilities and requirements via annotations

Requirements can only partially be enhanced through IDE support as the filter expressions still have to be written by hand. Therefore, we decided to only use `@Capability` and `@Attribute` annotations in our approach.

Capabilities, requirements and filters in combination with the OSGi resolver can be used to ensure that a system composed of bundles only is runable if all requirements are satisfied, thus preventing an incomplete system from being even started. This mechanism however is only applicable on bundle level. Therefore, it might be the case that the available bundles in the system provide all capabilities necessary, but the component within those bundles are referencing the wrong component and thus would lead to a malfunctioning system.

Hence, we also utilize another mechanism provided by OSGi that has already been introduced in subsection 4.3.1, i.e., properties and target filters of components and their references.

5.2.3. Component Property Types and Target Filters

Although, we already introduced the concepts of component properties and their target filters in subsection 4.3.1, we have to detail them as we will use some more advanced mechanisms than the already introduced ones in our following approach.

The `@Component` annotation is a compile time annotation that is used by external tools to detect for which `classes` a declarative services xml file should be generated in the `OSGI-INF` folder of a bundle. An xml file produced by such an external tool for a simple component with one property as depicted in Listing 5.5 is shown in Listing 5.6

```java
@Component(property="test=TEST")
public class ExampleComponent {}
```

Listing 5.5: Definition of a declarative service with a property

The amount of work needed for defining the component as a DS is drastically reduced as well as the chance to introduce an error into the xml. However, common components used in OSGi usually have more than one property, which rapidly can lead to complex and confusing lists of properties being declared within the `@Component` annotation's `property` property. Therefore, the OSGi R7 specification introduces a new concept called `ComponentPropertyType` that again reduces the amount of work needed to define a large amount of properties, thus reducing the risk of introducing an error and even providing a type-safe access of those properties that is supported by IDEs.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0" name="de.unia.
    smds.ExampleComponent">
  <property name="test" type="String" value="TEST"/>
  <implementation class="de.unia.smds.ExampleComponent"/>
</scr:component>
```

Listing 5.6: Definition of a declarative service with a property in xml

Again OSGi leverages the use of meta annotations in order to achieve this goal. The `ComponentPropertyType` annotation can be used to annotate other annotations that were so far used to describe configurations of components in a type-safe way. In Listing 5.7 we reused the previously introduced example of systems and games in order to exemplify the usage of `ComponentPropertyTypes` for components and references. The `ComponentPropertyType` annotation is applied to our self-defined `System` annotation. This annotation represents the configuration for a component, defining the properties `graphiccards` and `ram` which will be translated by an external tool into `system.graphiccards` and `system.ram`. The name of the annotation serves as namespace for the properties. We only used one word, i.e., `System` as annotation name, but given we used a camel-cased name like `SystemSomething` this would be translated into `system.something`. A component that is additionally annotated with the

`@System` annotation will automatically be registered in the OSGi registry with the respective values given for the two attributes, e.g., `HighEndSystem` will be registered with the `interface IHighEndSystem` and the properties `system.-graphiccards:List<String>="nvidia,onboard"` and `system.ram=32`. Another component, e.g., `HighEndGame` then can use these properties to filter for them within a `@Reference` target filter.

```java
@ComponentPropertyType
public @interface System{
    String[] graphiccards;
    long ram;
}

@Component
@SmdsCapSystem(
    graphiccards={"nvidia,onboard"},
    ram=32
)
public class HighEndSystem implements IHighEndSystem{}

public class HighEndGame {
    @Reference(target="(&(|(system.graphiccard=nvidia)(system.graphiccard=
        radeon))(system.ram>=16))")
    IHighEndSystem highEndSystem;
}
```

Listing 5.7: ComponentPropertyType annotated annotation applied to a component

This mechanism can be used to define complex property collections for a component that can be safely applied via annotations that are supported by IDEs.

	Cap-Req-Filter	**Property-Filter**
Level of Abstraction	Bundle	Component
Place of Definition	Manifest	service.xml
Used Mechanisms	Requirements Capabilities Filters	Properties Filters
Effect	Prevents Bundle Resolving	Prevents Component Resolving

Table 5.1.: Comparison between Capabilities-Requirements-Filter mechanism and Properties-Filter mechanism

Table 5.1 summarizes the differences between both presented mechanisms. Capabilities, requirements and filters are useful in order to prevent bundles form even starting or a system preventing from being assembled if not all needed parts are in place. Properties and filters can be used to prevent a complete system from malfunctioning due to components being assembled in the wrong way.

5.2.4. OSGi Registry and Proxies

Until now we presented two different concepts that can be used to restrict which bundles my be started in combination, i.e., requirements and capabilities, as well as to be used for fine-grained component selection during reference injection, i.e., `ComponentPropertyTypes`. What is missing is a mechanisms that allows us to also enforce specific rules for the interplay of components at runtime, i.e., not only at the time of reference injection but also for the time when the service of a component is actually used by another component.

This is where OSGi's Service Hook Service Specification [77] comes into play. These hooks enable arbitrary code to participate in the OSGi framework's service primitives: publish, find and bind. One capability of those hooks that is especially intriguing is the possibility to proxy another service transparently for a specific bundle. Or in general proxy all service calls and thereby adding cross-cutting functionality to all bundles in one place.

For this to work properly a bundle has to register a service that implements the `interfaces` `EventHook` and `FindHook`. A `EventHook` is called every time a service is registered, modified or unregistered, thus being the perfect place for adding proxies for those services to the system. A `FindHook` in contrast is called every time a service is requested, e.g., when a new component comes up and has a reference to another service. Therefore, this is the perfect place to hide existing services and replace them with their respective proxy. For creating arbitrary proxies, Java already offers several predefined concepts, see [78], like `InvocationHandlers` or the `Proxy` `class` with its static factory methods.

In order to illustrate the usage of those concepts we defined a proxy for all services in an OSGi runtime in Listing 5.8 that shows how to log every service invocation that is happening during runtime without changing the code of the service itself. Therefore, we defined a `class` `LogEventHook` which registers a proxy for all services that are registered. The `class` `LogFindHook` removes the proxied service from the find result so that another component will not see the service itself, but only the proxy which is added to the find results. Finally, an invocation of a method on the service is delegated to our proxy implementation `LoggingHandler` which implements the `InvocationHandler interface` from Java's `java.lang.reflect` package. This `InvocationHandler` is now called everytime a service is invoked and subsequently logs this method call via retrieving the called method's name via reflection.

```java
@Component
public class LogFindHook implements FindHook{
    ...

    @Override
    public void find(BundleContext bc, String name, String filter, boolean
        allServices, Collection references) {
        references.remove(proxiedService);
        references.add(proxy);
    }
}

@Component
public class LogEventHook implements EventHook{
    ...

    @Override
    public void event(ServiceEvent event, Collection contexts) {
        proxy = Proxy.newProxyInstance(classloader, interfaces, new
            LoggingHandler());
    }
}

public class LoggingHandler implements InvocationHandler {
    private static Logger LOGGER = LoggerFactory.getLogger(
        DynamicInvocationHandler.class);

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        LOGGER.info("Invoked method: {}", method.getName());
        return method.invoke(...);
    }
}
```

Listing 5.8: Simplified service proxying

5.3. Mapping

In this section we show how developers can benefit from the usage of OSGi in a generated, modular code base with regard to enforce contracts that go beyond the syntactical level of simple `interfaces` and their declared types for method parameters and their return values. First, we will introduce several capabilities that serve as examples for what can be done with OSGi's requirement and capability model, accompanied by corresponding component property types. The defined capabilities serve as contracts at buildtime, while the component property types are used at runtime during the dependency injection phase for each component. Finally, we will show how the contracts defined by requirements, capabilities and component property types can be leveraged by a central component to enforce those contracts at runtime throughout each service invocation. This will be done by leveraging OSGi's service hooks and Java's proxy mechanisms.

5.3.1. Compile Time and Dependency Injection Time Contracts

In order to enable developers of SCSS to easily show the semantic equivalence of two or more components within a system, we will provide four different capabilities and their corresponding properties that will leverage OSGi's resolver in order to automatically show satisfaction or rejection of a component or bundle intended to be a substitute for another one. For bundles those capabilities can be used to enforce semantic equivalence as guaranteed by a developer at compile time, while the corresponding component property types enforce this contract at runtime, more precisely at dependency injection time, i.e., whenever a reference to another service is injected. In the following we use the word "capability" in order to describe both, capability and its corresponding properties.

The four capabilities target different scenarios that can be mapped onto a general requirements, capabilities, properties and filtering system as presented in subsection 5.2.2. The first capability targets type ranges for primitive types that are used as parameters or return values in methods of components. A range of values can be specified that is accepted by the respective method. The second capability can be regarded as an extension of the first one and maps pre- and post-conditions that can be given for a specific method. This enables developers to define a range of states within which their method will work as intended as well as the possible range of outcomes that can be expected by a caller. The third capability targets functional requirements as jitter and latency. This capability defines the maximum amount of time a method may execute and a range within which this execution time may vary. Finally, we present a hardware capability that complements the aforementioned one as usually the time a method needs to execute depends on the hardware it is running on.

Those four capabilities are explained in detail in the following subsections. For each capability we first define the namespace and its attributes, followed by a detailed example on how to apply the capability to either a method or a bundle.

5.3.1.1. Value Range Capability

Components of a SCSS often represent sensors, actuators or other parts of the real hardware of the system. Due to physical restrictions those parts often only work under specific conditions, e.g., an actuator only works within a specific temperature range as, if it gets too hot or too cold, the mechanical parts run the risk to malfunction or even get irrevocably damaged. Especially a damage to a non-reachable embedded system, e.g., a satellite or spacecraft, would pose a significant risk, be it financial or safety-critical in terms of human lives.

Thus, we will offer a capability that enables developers of such components to specify ranges of values they are able to work with, e.g., a specific temperature range, whereby we only support primitive types, due to restrictions of OSGi and because complex types or any sort of collection would go beyond the scope of this work.

```
de.unia.smds;
    de.unia.smds=typerange;
        class.interface:String;
        class.method:String;
        parameter.x.range.minimum[.excluding]:double;
        parameter.x.range.maximum[.excluding]:double;
        parameter.x.regex:String;
```

Listing 5.9: Definition of a type range capability

In Listing 5.9 we provided a capability that can be leveraged to describe the allowed type range of method parameters, thus enabling a developer to restrict the usage of this method to a limited set of possible input parameter values. The capability is defined within the namespace de.unia.smds, with name typerange and provides five different attributes.

The first one, i.e., parameter.interface, is of type String and together with class.method, also of type String, is used to unambiguously define the interface of a component and method to which this capability belongs to. class.interface for example would be the fully qualified class name, e.g., de.unia.-smds.contracts.RestrictedTypesInterface, whereas class.method would reference the name of the respective method within this class, e.g., restricted-TypesMethod. RestrictedTypesInterface is one of possibly several interfaces a class within the bundle implements. We chose to rather use the interface instead of a class as otherwise we would leak implementation details to other bundles. Therefore, we adhere to the best practices of OSGi which registers implementation classes with their respective interfaces and only expose one of those interfaces to other bundles.

The other three attributes, i.e., parameter.x.range.minimum, paramter.x.range-.maximum and paramter.x.regex, can then be used to describe the different primitive parameters of a method. Depending on the type of each parameter either the use of paramter.x.range.minimum and paramter.x.range.maximum is appropriate, i.e., when describing the possible values of a numerical paramter, e.g., long or float, or the use paramter.x.regex is appropriate in order to describe the possible values of a String or char parameter.

```
Bundle-Symbolic-Name: RestrictedTypesBundle
Provide-Capability:
    de.unia.smds;
        de.unia.smds=typerange;
            class.interface:String=de.unia.smds.contracts.
                RestrictedTypesInterface;
            class.method:String=restrictedTypesMethod;
            parameter.0.regex:String=[0-9A-Z]*;
            parameter.1.range.minimum:double=0;
            parameter.1.range.maximum:double=20;
```

Listing 5.10: Type range capabilities within bundle manifest

Per default paramter.x.range.minimum and paramter.x.range.maximum are including if not explicitly defined otherwise by extending this attribute to parameter.x.range.minimum.excluding or paramter.x.range.maximum.excluding respectively. The x within the attribute names refers to the position of a parameter within the method signature starting at 0, e.g., given a method restrictedTypes-Method(String input1, Long input2), then the typerange attribute for input1 is parameter.0.regex:String. There is no explicit attribute for boolean values, as there is no sense in restricting a binary parameter. In case of a `boolean` parameter being restricted, the code of the method per default can assume this value to be either `true` or `false`, but has not to explicitly define this fact.

An application of this capability within a bundle manifest can be seen in Listing 5.10. Within the manifest a `Provide-Capability` header is declared with our capability and the parameters set to match the `interface` and its implementation shown in Listing 5.11. The first parameter `input1` is defined to only accept inputs in form of an arbitrary combination of numbers and upper letter characters and the second parameter `input2` only accepts values between 0 and 20, whereby both are included.

```java
public interface RestrictedTypesInterface {
    public void restrictedTypesMethod(String input1, long input2);
}

@Component
public class RestrictedTypesImpl implements RestrictedTypesInterface {
    @Override
    public void restrictedTypesMethod(String input1, long input2){ ... }
}
```

Listing 5.11: Interface and implementation for restricted types

As already mentioned in section 5.2, by using capabilities and requirements a developer can only express contracts on a bundle level between single bundles. This way a system still is able to be set up wrong, i.e., components that reference the wrong service, although the right one is provided by the system. Therefore, we also defined the corresponding properties for components in order for those to be referenceable by other components.

In Listing 5.12 we depicted two `ComponentPropertyType` annotated annotations, i.e., `TyperangeNumeric` and `TyperangeString`, that can be used to annotate components that want to restrict service method parameter values. As their bundle level counterpart capability `TyperangeNumeric` and `TyperangeString` they define virtually the same attributes, i.e., `method` for the method name, `param` which is the same as `x` in the capability and the respective attributes to define the range of each parameter. The attribute `class.type` is omitted, as it is implicitly given by the component itself which implements the `interface` that is normally referenced by this attribute. If a component is annotated with this `ComponentPropertyType` then other components can reference it via defining a target filter for the corresponding `@Reference` annotation.

The application of this annotation is also shown in Listing 5.12. The `class`

RestrictedTypesImpl is annotated with TyperangeString and TyperangeNumeric in order to restrict the parameters of its service interface to be an arbitrary sequence of numbers and upper letter characters for input1 and a double between 0.0 and 20.0 for input2.

```
@ComponentPropertyType
public @interface TyperangeNumeric{
    String method();
    long param();
    double max();
    double min();
    exclude_min() default true;
    exclude_max() default true;
}

@ComponentPropertyType
public @interface TyperangeString{
    String method();
    long param();
    String regex();
}

@Component
@TyperangeString(method="restrictedTypesMethod", param=0, regex="[0-9A-Z]*")
@TyperangeNumeric(method="restrictedTypesMethod", param=1, min=0.0, max=20.0)
public class RestrictedTypesImpl implements RestrictedTypesInterface {
    @Override
    public void restrictedTypesMethod(String input1, long input2){ ... }
}

@Component
public class UserOfRestrictedTypes {
    @Reference(target="(
        &
        (typerange.numeric.method=restrictedTypesMethod)
        (typerange.numeric.param=0)
        (typerange.numeric.regex=[0-9A-Z]*)
        ...
    )")
    RestrictedTypesInterface restricted;
}
```

Listing 5.12: Definition of type range property annotations and their usage

Those properties are then used by UserOfRestrictedTypes to reference an implementation of RestrictedTypesInterface with those specific restrictions via a target filter which is only shown in a shortened form here, i.e., only targeting the first parameter. This way several implementations of RestrictedTypesInterface can be present in a running system, each having different restrictions regarding the parameter values they can work with, while a developer can rest assured that he only gets the implementation that adheres to his specific restriction requirements. If there is no such implementation, OSGi could not satisfy the requirements of the component and therefore the component would not be able to start. Thus, there is no way to ever run a system that might malfunction due to inappropriately calling methods with wrong parameter values.

5.3.1.2. Pre-/Post-Condition Capability

As already mentioned in subsubsection 5.3.1.1, components of a SCSS often represent sensors, actuators or other parts of the real hardware of the system and therefore, have to to specify ranges of values they are able to work with, e.g., a specific temperature range, due to physical restrictions of those parts.

In subsubsection 5.3.1.1 we have shown how such a restriction might look like for parameters of methods. However, not only the parameters of a method are of interest for a developer of a SCSS but also its return value. Thus, in this subsection we enhance the capability presented in subsubsection 5.3.1.1 with additional attributes that can be used to restrict the return value of a method. By adding this additional attribute we partially enable a pre-/post-condition semantic for methods which is an already field-proven methodology to enforce semantic restrictions without declaring the inner workings of a method. However, our approach is not only usable for methods, but also for describing sort of pre-/post-conditions on a bundle level.

```
return.regex:String;
return.range.minimum[.excluding]:double;
return.range.maximum[.excluding]:double;
```

Listing 5.13: Return range attributes within type range capability

As our new capability is merely an enhancement of our previous capability, we will only depict the additions we have made instead of defining the whole capability again. This means we are reusing the namespace `de.unia.smds` as well as the capability name `typerange` of our previous capability.

In Listing 5.13 we therefore, depicted only the additional attributes `return.regex` of type `String` and the corresponding attributes for numerical values, i.e., `return-.range.minimum` and `return.range.maximum`, both of type `long`. Those attributes are used exactly the same way as their parameter counterparts defined in subsubsection 5.3.1.1, i.e., defining either a regex that denotes the form of a returned `String` or the range of returned numerical value, per default including the minimum and maximum value. The `x` part of the parameter capabilities was omitted as there is only one return value for Java methods.

```
Bundle-Symbolic-Name: RestrictedTypesBundle
Provide-Capability:
    de.unia.smds;
        de.unia.smds=typerange;
            ...
            return.regex:String=[0-9]*;
```

Listing 5.14: Return range attribute defined within bundle manifest

As for our previous capability, we also define the corresponding properties that can be used in conjunction with components and references. The definition of return range properties is shown in Listing 5.15. As well as their capability counterpart those look very similar to the type range property types shown in Listing 5.12, apart from the fact that `ReturnrangeNumeric` and `ReturnrangeString` both do not define a `parameter` attribute. This is due to the same reason mentioned before: Java methods only have one return value.

```
@ComponentPropertyType
public @interface ReturnrangeNumeric{
    String method();
    double max();
    double min();
    exclude_min() default true;
    exclude_max() default true;
}

@ComponentPropertyType
public @interface ReturnrangeString{
    String method();
    String regex();
}
```

Listing 5.15: Definition of return range property annotations

The usage of these `ComponentPropertyTypes` is the same es already shown in Listing 5.12, i.e., they can be used to annotate a component implementing a service `interface` and then be referenceable by another component through the corresponding `@Reference`'s target filter.

By using this enhanced version of type ranges for both, bundles and components, a developer is now able to define proper pre- and post-conditions for each method defined in a service `interface` on component and bundle level. Currently, this approach is restricted to primitive types, i.e., `String` and numerical values.

5.3.1.3. Timing Capability

Another functional requirement, besides accepting and returning only specific value ranges, that is often posed on SCSS, is a response within a given time. This sort of requirements falls into the category of real-time requirements and have been introduced in subsection 2.3.2 as latency and jitter. Latency has been defined as *"[...] a measure of time between a particular event and a system's response to that event [...]"* [24] and jitter as *"[...] the variation or unsteadiness in a measured quantity."* [24].

In our current context this means the time a specific method takes to execute and return results as well as the variation of this time, e.g., the aforementioned method `restrictedTypesMethod` might take 10ms to execute and this time might vary between 8ms and 12ms.

For the capability defined in the following subsection we decided to only use Latency as an attribute, but in a special form: WCET. Effectively, this is the utmost latency value measured for an execution of a method, usually added with an additional threshold, e.g., in case of `restrictedTypesMethod` this would be 12ms plus an additional threshold of 2ms resulting in a WCET of 14ms. The threshold usually is either an empirical value derived from experiments or a captured value for similar methods.

```
Bundle-Symbolic-Name: WCETBundle
Provide-Capability:
    de.unia.smds;
        de.unia.smds=wcet;
            interface:String;
            method:String;
            wcet:long;
```

Listing 5.16: Definition of a WCET capability

The WCET capability is shown in Listing 5.16. Similar to the typerange capability the WCET capability is defined within the namespace `de.unia.smds` but with `wcet` as name instead of `typerange`. Also similar to the typreange capability we need an unambiguous way to define the `interface` and the method for which a WCET capability is provided, thus we have to define the two attributes `interface` and `method` that are used the same way as in the typerange capability. The last attribute, i.e, `wcet` finally defines the WCET for this method in milliseconds.

As the application of this capability follows the same rules as shown in the previous subsections we refrain from showing it again, but directly define the corresponding `ComponentPropertyType` definition.

```java
@ComponentPropertyType
public @interface WCET{
    String method();
    double wcet();
}
```

Listing 5.17: Definition of a WCET property annotation

This definition is shown in Listing 5.17. The attribute `method` of type `String` is used as its capability counterpart to define the method this WCET applies to. The WCET is given by the eponymous attribute `wcet` which is of type double and is used to define the WCET in milliseconds. The name of the `interface` is again given implicitly, as the `WCET` annotation will be used to annotate a component that implements a service `interface`.

This WCET capability can now be used by developers to even further restrict the semantics of methods on component and bundle level. Given the requirements of a hard real-time guarantee then a bundle or component can require the system to provide another bundle or component to provide a specific method not only with restricted type ranges but also to always answer within specific time bounds. This capability will be supplemented with our last capability in the next subsection, which deals with hardware restrictions a component or bundle can pose to a system they are running in.

5.3.1.4. Hardware Capability

The last of our four capabilities deals with hardware requirements a bundle or component can have. Those hardware requirements must be satisfied by the system the bundles or components are running in, e.g., a bundle is only able to run on a specific processor architecture, because it uses native libraries that are compiled only against x86, or it requires a minimum amount of RAM in order to be operable. The list of possible hardware requirements can be rather exhaustive, therefore we restricted our approach to only address the most common ones, i.e., RAM, CPU frequency, architecture, disk. Those are features that are usually offered by any platform. Others might be specific sensors or actuators, graphiccards or external conditions, as available bandwith for an internet connection.

```
Bundle-Symbolic-Name: HardwareBundle
Provide-Capability:
    de.unia.smds;
        de.unia.smds=hardware;
            architecture:String;
            cpufreq:long;
            ram:long;
            disk:long;
```

Listing 5.18: Definition of a hardware capability

As out other capabilities before, the hardware capability defined in Listing 5.18 uses `de.unia.smds` as namespace and defines its own name within it, i.e., `hardware`. The four properties target the aforementioned general properties of a system. `architecture` describes the architecture of a system, e.g., x86 or ARMv7, in form of a `String`. `cpufreq` defines the frequency the CPU of the system works at in Hertz, e.g., 1000000 would describe a CPU with frequency 1MHz. This excludes all CPUs working at a lower frequency than 1Hz, but we believe that this is sufficient for the vast majority of systems out there today. `ram` is used to describe the amount of RAM a system has in terms of Byte, e.g., 1024 would define a system to have one KByte. Finally, `disk` is used to also express the amount of memory, but this time for disk memory, which is usually larger than RAM. It is also given in Byte.

Although, the hardware capability is not bound to a specific method or `interface` it can still be used at component level to describe components that represent the system a SCSS is running on. Therefore, we also defined a `ComponentPropertyType` for hardware, as can be seen in Listing 5.19.

```
@ComponentPropertyType
public @interface Hardware{
    String architecture();
    long cpufreq();
    long ram();
    long disk();
}
```

Listing 5.19: Definition of a hardware property annotation

The previously described attributes of Listing 5.18 are repeated in Listing 5.19. As there is no coupling to an `interface` it has not be implicitly derived by the `interface` of the `class` `Hardware` is annotated to.

The capability and `ComponentPropertyType` defined in this subsection can be used by a developer to unambiguously describe the capabilities of the system the SCSS is running on. Thus, bundles and components can define requirements on hardware level that have to be satisfied in order to run them appropriately. This eliminates the risk of running a component or a bundle on hardware that is not fit to run it. This could be the case for a too slow processor that is not able to execute all methods within their defined WCET and therefore will break the scheduling that would be working on an appropriate machine. Also out of memory situations could be handled by this capability by exactly defining how much RAM and disk space bundles and components expect to be available.

5.3.2. Runtime Contract Enforcement

In the former subsection we illustrated several capabilities together with their corresponding component property types that enable a general contract definition for different use-cases, e.g., type value ranges, pre-/post-conditions and even hardware dependencies and WCET for methods.

However, until now all those contracts rely on the thoroughness of a developer who has to write those contracts by hand. No third party is involved yet that ensures the compliance of the written or generated code with the stated values of the contract. Therefore, in this subsection we propose an additional runtime component that is able to check some of the above named capabilities at runtime, thus enforcing the compliance of deployed code with contracts that they are bound to.

This component makes heavy use of OSGi's service hooks and Java's proxying mechanisms, see subsection 5.2.4. For the sake of brevity, the component will only support `String` type value ranges, but could be easily adapted to also support pre- and post-conditions or other types of contracts defined by a developer. In order to enforce value range types at runtime we will have to proxy all services that declare such a contract. Therefore, we have to register an `EventHook` that takes care of registering a proxy for each of those services. In Listing 5.20 we depicted the `event` method of such an `EventHook`, which is called before event delivery, i.e., registered, unregistered, modified. First, we check if the service even needs proxying by looking for the property `method` which should be present if the respective component property type annotation was used at compile time. If that is the case, then we make sure this service is not already proxied, thus preventing an infinite proxy loop. Finally, we extract the properties relevant to the proxy, i.e., `param`, denoting the position of the parameter within the method and `regex` which defines all possible values this parameter might represent. Those are subsequently passed to the method that creates the proxy for this service.

```java
@Override
public void event(ServiceEvent event, Collection contexts) {
    final ServiceReference serviceReference = event.getServiceReference();
    String method = serviceReference.getProperty("method");
    boolean isProxied = serviceReference.getProperty("isProxied");
    if (method = null && !isProxied) {
        Bundle bundle = serviceReference.getBundle();
        switch (event.getType()) {
            case REGISTERED: {
                String regex = serviceReference.getProperty("regex");
                int parameter = serviceReference.getProperty("param");
                String[] propertyKeys = serviceReference.getPropertyKeys();
                Properties properties = buildProps(propertyKeys, event);
                String[] interfaces = (String[]) serviceReference.getProperty(
                    "objectClass");
                Class[] toClass = toClass(interfaces, bundle);
                proxyService(bundle, toClass, properties, this.getClass().
                    getClassLoader(), new StringContractProxy(bc,
                    serviceReference, method, parameter, regex));

                break;
            }
            ...
        }
    }
}
```

Listing 5.20: Scetched EventHook for a contract enforcement component

In Listing 5.21 we depicted the proxy that is actually enforcing the contract, i.e., `StringContractProxy`. First, the regex given to this proxy is compiled to a regex pattern which then is used during method invocation to check if the passed string adheres to the regex defined withing the contract. If it matches then the actual service is called with the given value and the proxy returns the same value as the original method would have. Otherwise, a message is printed stating a contract violation has happened and a `null` value is returned. This is a rather simplistic default behavior, but could easily be adapted to be a more sophisticated, centralized contract violation handling facility.

```java
public class StringContractProxy implements InvocationHandler{
    ...
    private Pattern pattern;

    public StringContractProxy( ... String method, int parameter, String regex
        ) {
        ....
        this.pattern = Pattern.compile(regex);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        Object invoke = null;
        if(pattern.matcher(args[parameter]).matches())
            invoke = method.invoke(bundleContext.getService(serviceReference),
                 args);
        else
            System.out.println("Contract Violation!");
        return invoke;
    }
}
```

Listing 5.21: StringContractProxy enforcing contracts at runtime

In order to force all components in a system to really use our proxy, at least in sensible cases, we finally have to define our own `FindHook` that intercepts all calls to the OSGi registry whenever a bundle or a component asks for references to a specific service. In Listing 5.22 we depicted such a `FindHook` implementation. This implementation is somewhat simplified as it merely checks if the property `isProxied` is set to false and if so removes this service from the returned list of possible target services. Given the case that more than one type of proxy is deployed to a system, then this implementation might leave them in the returned list too. The implementation also ignores the target filter entirely. However, in our small example this `FindHook` implementation would only return proxied services which subsequently would enforce the contracts posed on theses services.

```
@Override
public void find(BundleContext bc, String name, String filter, boolean
    allServices, Collection references) {
    Iterator iterator = references.iterator();
    while (iterator.hasNext()) {
        ServiceReference sr = (ServiceReference) iterator.next();
        if (sr.getProperty("isProxied")) {
            iterator.remove();
        }
    }
}
```

Listing 5.22: FindHook find method implementation removing all non-proxies

5.4. Related Work

In this section we first examine approaches that are also promoting contracts for component-based design in general, followed by approaches that are used in practice for Java as it is our target language. Finally, we demarcate our approach from OSGi contracts, which are an already standing concept but do not fulfill the requirements we had for our definition of contracts.

5.4.1. General Contract Concepts

In [79] an approach is proposed to write contracts for components either in form of pre- and post-conditions or in the form of interface state machines via the Abstract State Machine Language that is proposed by the authors. The Abstract State Machine Language is based on Abstract State Machines as defined by [80] and enables the full behavioral description of a component. The approach targets the .NET runtime, i.e., languages like C#, VisualBasic.NET and many other languages running on top of the Common Language Runtime. Instead of static checking during compile time the authors of [79] intend to check contracts during runtime by using monitors that check a specific implementation against its specification, whereas our approach targets both, compile time checks so that incompatible bundles can not be started together and also runtime checks, enabled by OSGi that ensure a specific component is not started as long as its requirements are not satisfied and given the case it is started its contract is dynamically monitored. Another difference is the targeted language or virtual machine. [79] targets the Common Language Runtime, whereas our approach is targeting the JVM and specifically OSGi on top of it. Although, [79] therefore targets a broader range of languages, where we only target Java, [79] has no default mechanisms for describing components other than as methods or objects in general, where we rely on OSGi as a rather restrictive framework, specifically aiming at component-driven development.

[81] and [82] both propagate a similar approach, i.e., the annotation of `interfaces` in Java to enforce semantic contracts between implementation of an `interface` and user of an `interface`, rather than solely relying on syntactical contracts posed by the `interface` itself. The annotations of both approaches slightly differ from each other. [81] proposes comment-style `@Pre`, `@Post` and `@Invariant` annotations that have to be declared within a comment of a method declaration

in an `interface` (or a `class`) as shown in Listing 5.23. Those – sort of – annotations are processed by an external tool and are translated into source code which, at runtime, checks if the respective pre-/post-conditions and invariants hold true during execution. In case of a violation of the contract an exception is thrown.

```java
public interface Person{
    /**
    * @post return > 0
    **/
    int getAge();

    /**
    * @pre age > 0
    * @invariant age > 0
    **/
    void setAge(int age);
}
```

Listing 5.23: Definition of a contract as defined in [81]

[82] proposes real annotations similar to those presented in Listing 5.23 but with an additional `@Contract` annotation that has to be present on an `interface` in order to annotate its methods with either `@Pre`, `@Post` or `@Invariant`. The rest of the process is similar to the approach of [81], although IDE support of [82] is partially better as they are using annotations that a compiler can understand. In contrast to our approach, both approaches target only runtime and only pre-/post-conditions, where our approach targets also compile time checks and can be applied to a broader range of possible contracts.

Both, [82] and [81], are rather old approaches dating back to 1998 and 2006 respectively. Current approaches that are used in practice are mainly found in tests rather than production code, e.g., AssertJ [83]. AssertJ is mainly used in unit testing to assert a specific state, i.e., a precondition, before the test starts. At the end of a test other assertions can be used to define the state at the end of the test, i.e., post-conditions. If used during a test, assertions can be seen as invariants that have to hold true during each invocation of a test. Although, AssertJ can also be used to assert specific conditions at runtime, developers refrain from using the library this way, as it encompasses a non-neglectable impact on runtime performance. In contrast to our approach, AssertJ is mainly used during integration or unit tests, whereas our approach specifically targets compile time and runtime checks.

In the context of microservices and their often REST-oriented interface design, contract solutions like Pact [84] emerged during the last years.

> *"Pact is a contract testing tool. Contract testing is a way to ensure that services (such as an API provider and a client) can communicate with each other."*[84]

Pact provides a Domain-Specific Language (DSL) to define consumer driven contracts, i.e., contracts that only test parts of the communication that are actually used by the consumer. Contracts that are written with Pact DSL are registered with a mock service. A consumer then fires a request to a mock provider which compares this request with an expected request (as defined by the contract). Given the request is valid the mock provider returns a minimal expected response (as defined by the contract) which can then be verified by the consumer. In contrast to our approach Pact usually is used in unit tests to verify the part of a REST API consumers interact with, where our approach targets compile time and runtime checks. Also, Pact only can be used to verify contracts that are implemented as REST interfaces, where our approach, although not technology independent, is not bound to any specific interface technology.

5.4.2. OSGi Contracts

A system in OSGi usually consists of a number of bundles, each encapsulating a specific functionality. Each of of those bundles usually declares services that shall be registered in the OSGi registry and usually those services depend on each other. Those dependencies are reflected within the bundle manifest by a specific header called *Import-Packages*, where the packages of those service are listed that the services of the respective bundle depend on. Such a header is depicted in Listing 5.24, where `bundleA` depends on `classes` or `interfaces` defined in the package `de.unia.smds.exported.package`.

```
Bundle-Symbolic-Name: bundleA
...
Import-Packages: de.unia.smds.exported.package
...
```

Listing 5.24: Import-Packages header in bundle manifest

Another bundle `bundleB` exports exactly this package via declaring another header, i.e., `Export-Packages`, in its manifest file, as depicted in Listing 5.25. When `bundleA` is installed into an OSGi system, then a resolver tries to resolve those dependencies before starting `bundleA`. If the dependency can be resolved, then bundleA is started, if not, then it stays in the Installed state until bundleB is installed in the system too.

```
Bundle-Symbolic-Name: bundleB
...
Export-Packages: de.unia.smds.exported.package
...
```

Listing 5.25: Export-Packages header in bundle manifest

In addition to package names a bundle depends on, each package can have a version range, e.g., [1.0.0,2.0.0) meaning all versions including 1.0.0 and excluding 2.0.0. Those versions are based on semantic versioning [75], which exactly defines when to increment major, minor or macro part of the versioning scheme, thus enabling an automatic baselining approach if all bundles of a system adhere to this versioning scheme. Unfortunately, the Java platform itself does not version its packages and therefore breaks OSGi's mechanism of automatic resolving and baselining. Therefore, OSGi itself has introduced so called Portable Java Contract Definitions (PJCD) [85]. This is important because those contracts are not what we will utilize for our approach, but could easily be mistaken as an alternative approach based on the naming alone. OSGi's PJCD deal with the problem introduced by Java platform bundles that do not adhere to the rules of semantic versioning and thus breaking the above explained mechanism of fine grained import and export dependencies between bundles.

Therefore, PJCD enabled a developer to depend not only on exported packages of another bundle, but also on arbitrary contracts. So instead of listing all packages a bundle depends on, it is now also sufficient to only depend on one specific attribute of another bundle. In Listing 5.26 this is exemplarily depicted for bundleC, which now provides a general osgi.contract capability Servlet in version 3.0., but not all the different packages that usually come with this specification and which are usually not listed by a Java platform bundle. A bundle depending on the Servlet capability can now easily declared this by adding a corresponding Require-Capability header in its manifest.

```
Bundle-Symbolic-Name: bundleC
...
Provide-Capability:
  osgi.contract;
    osgi.contract=Servlet;
    version="3.0"
```

Listing 5.26: Contract in bundle manifest

In order to enable PJCD the same mechanism also import and export headers are based on was leveraged: requirements and capabilities. Although this is the same mechanism our approach is based on, OSGi contracts target a completely different use-case, i.e., wrong defined dependencies.

5.5. Evaluation

For the evaluation of our approach we reused the running example presented in section 2.1. As we already compared aadl2osgi against aadl2rtsj and the golden standard in the former chapter, we here only compare our new approach against aadl2osgi. As this approach does not add new mapping concepts regarding code generation, but only handwritten code and additional runtime components we leave out the comparisons of generated code size as those might vary significantly from use-case to use-case and therefore a general comparison would make no sense. However, we again examine the throughput of our approach compared against a plain aadl2osgi solution, as well as we provide a qualitative evaluation on different aspects of our approach for scenarios like adding or replacing components.

5.5.1. Quantitative Evaluation

In order to evaluate our contract approach we deployed two different contract enforcement components into our system. Both resemble in structure the example described in subsection 5.3.2, i.e., consist of an `EventHook`, `FindHook` and an `InvocationHandler` implementation which serves as a contract proxy. The first proxy is shown in Listing 5.27 and serves as a proxy for methods that have numbers, more precisely integer numbers, as parameters.

```
public class NumberContractProxy implements InvocationHandler{
    ...
    private int upperBound;
    private int lowerBound;

    public NumberContractProxy( ... int upperBound, int lowerBound) { ... }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        Object invoke = null;
        int value = args[parameter];
        if(lowerBound <= value && upperBound >= value)
            invoke = method.invoke(...);
        else
            System.out.println("Contract Violation!");
        return invoke;
    }
}
```

Listing 5.27: NumberContractProxy enforcing contracts at runtime

The proxy is given an `upperBound` and a `lowerBound`, both inclusive, that determine the value range a given parameter may be in. Given the case a parameter is within this range the method invocation is executed, otherwise a message is printed. This structure is similar to the one of `StringContractProxy` used in subsection 5.3.2 which also is our second proxy under consideration.

For both contracts we ran message throughput benchmarks, similar to those executed in subsection 4.5.1 in order to determine their impact on message throughput. The results of those benchmarks can be seen in Figure 5.2.

In this figure we depicted three different calldepths, i.e., 1, 5 and 10 for three different settings. The first setting is a normal setup as already used in subsection 4.5.1 with no contract enforcement in place, thus serving as a reference we can compare our contract enforced solution against. The results of this setting are labeled 1, 5 and 10 respectively. The second setting, i.e., with our above shown `NumberContractProxy` in place, is labeled as n1, n5 and n10 while the numbers again denote the calldepth. Finally, the last setting, i.e., with a `StringContractProxy` in place instead of a NumberContractProxy, is labeled as s1, s5 and s10 accordingly.

To no surprise the application of runtime contract enforcement comes at a cost materializing in a lower throughput of both, `NumberContractProxy` and `StringContractProxy` setting. However, the throughput of our `NumberContractProxy` still is significantly higher than the one of `StringContractProxy`. This difference in throughput probably stems from regular expression pattern matching being a more costly operation than a primitive greater/smaller than comparison. Nevertheless, both `NumberContractProxy` and `StringContractProxy` suffer from an increased calldepth as the origin aadl2osgi solution already did. Whereby, `StringContractProxy` seems to suffer less than `NumberContractProxy`. This difference though we can not explain and thus ascribe it to under-the-hood optimizations of the JVM regarding strings.

It remains to be said that in general number comparisons tend to be faster than string comparisons. Hereby, a solution with number contracts, depending on the calldepth, runs at 1/3 the speed of a solution without contract enforcement, where a string contract solution only runs at 1/15 of the speed of a solution with no contract enforcement.

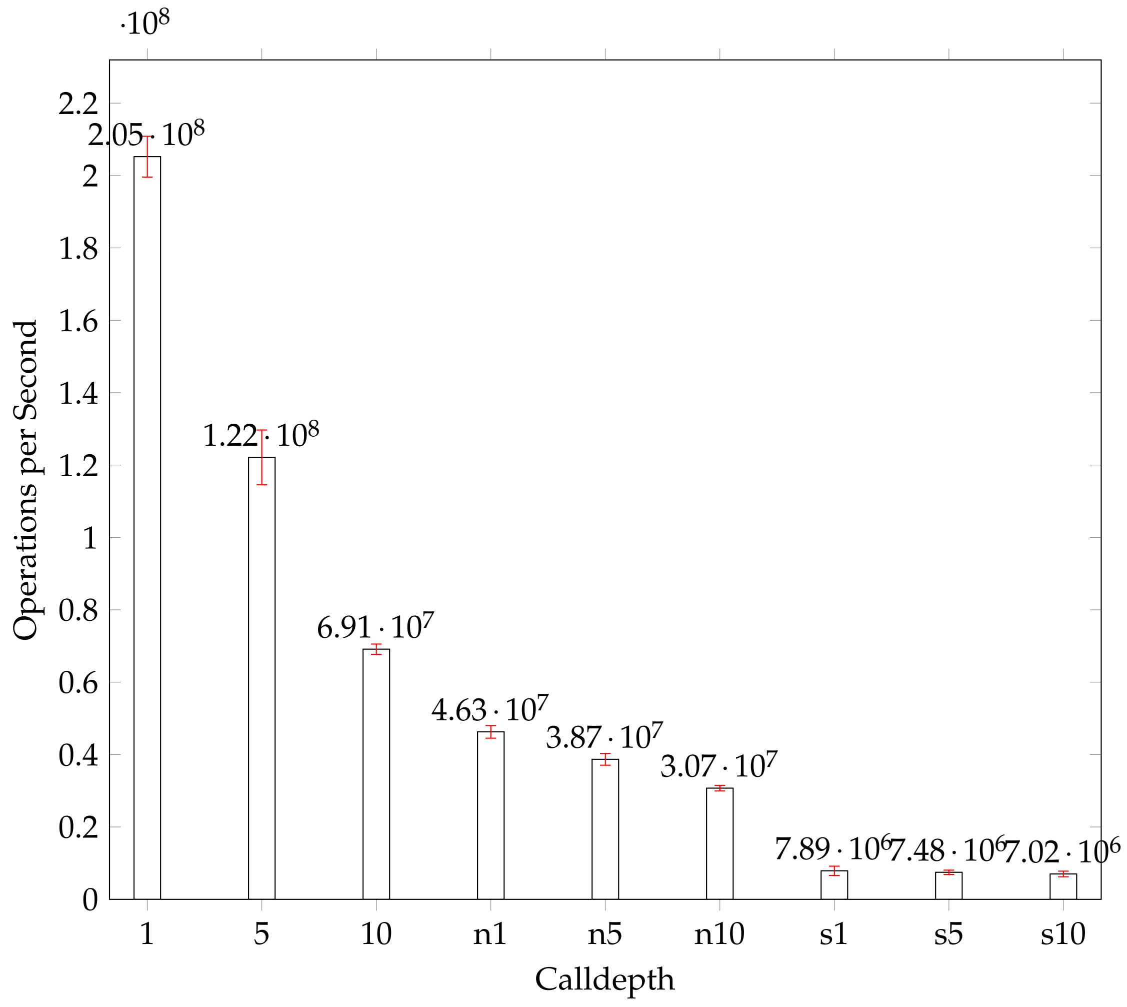

Figure 5.2.: Throughput aadl2osgi without/with number/with string contract enforcement and different calldepths

5.5.2. Qualitative Evaluation

Aside from a purely quantitative evaluation we also played through different scenarios that put the contract based solution in contrast to the generated, modular solution in terms of restrictiveness and semantic equivalence. We will again reuse the running example defined in section 2.1.

Functional/Non-Functional Requirements For our first use-case we have chosen `PitchControl`, `RollControl` and `YawControl` to serve as example components that can benefit from the additional possibilities offered by our contract based approach presented in this chapter.

All three control one degree of freedom of the quadrocopter. `PitchControl` controls how much the quadrocopter needs to tilt in the direction of its movement or the opposite direction and thus usually controls the speed of the quadrocopter. `RollControl` does the same but for the directions shifted 90° to the movement direction. Finally, `YawControl` controls the direction in which the quadrocopter is looking by turning it around the z-axis. In order to work properly all of them need information about the current position within space of the quadrocopter. In our running example presented in section 2.1 this information is provided by a simulation. In reality such information usually is given by sensor hardware, e.g., a gyroscope. Therefore, in a real-world setting our three controllers would need a gyroscope to be present in order to work properly. Such a requirement we are now able to express via a capability that developers can define. For the sake of simplicity we assume the hardware capability presented in subsubsection 5.3.1.4 also has defined an attribute for sensors that needs to be present. In such a setting our three controllers each could use such an attribute to express their need for a gyroscope sensor to be present in order to function properly. Otherwise the system would be prevented from even starting, which, considering an autonomous flying quadrocopter with no sense of its position whatsoever, might be just the result we would want from the very beginning.

Another requirement could be the unit of measurement used by the components internally. Usually it is not a good idea to combine several modules to solve complex computations when some of those modules use other units of measurement for physical entities than others. This is vividly shown by [86], which mentions *"[...] thruster performance data in English units instead of metric units [...] in the software application code [...]"* as one of the root causes of the disastrous loss of Mars Climate Orbiter in 1999.

In comparison, within the setting of our former approach we weren't able to express such requirements at all. Therefore, a quadrocopter with no gyroscope would have been successfully started and, once airborne, probably eventually crashed. Thus, our approach obviously is a better fit for use-cases where software needs to pose requirements that target its surrounding hardware or other functional/non-functional requirements that could not be expressed directly in code, than an approach where those expressions are not possible.

Semantic Equivalence Aside from preventing components from starting when their surroundings do not meet their requirements, we also explained how a centralized component can enforce such requirements at runtime, i.e., after a component is started. For this use-case we chose the **`AltitudeController`** to serve as an example component. Given the case an implementation of **`Altitude-Controller`** only works within a certain value range, e.g., 0 - 1000 meter, then we could deploy an external component as shown in Listing 5.27 to enforce this contract at runtime.

In such a setup the implementation of **`AltitdueController`** either is given the right values to work with, i.e., values between 0 and 1000, or the central `Number-ContractProxy` signals a contract violation which might result in a graceful system shutdown, i.e., landing the quadrocopter immediately. The point is, the behavior for a contract violation is defined within an external component that monitors each component that claims to adhere to this contract. Now, given the case a component violates the contract the system is still able to react to such cases, e.g., via shutdown or replacing the malicious component. If such an external component is written and certified once, this would greatly enhance the certification of new components that are monitored by this component.

First, the contract stated by a developer is checked at system startup and, if the component does not resolve, prevents it from even starting. Secondly, this component is proven to work correctly by the (certified) runtime monitor which is able to prevent the component from doing harm to the system. Which should assist developers at showing semantic equivalence of different components.

5.6. Conclusion

In this section we presented an extension to our former approach aadl2osgi, targeting not only a modular code base by design but also a concept to express dependencies between modules and their surrounding hardware as well as other contracts. This approach enables developers of a SCSS to not only shift structure, timing and communication-related concerns into design phase and to create a highly modular, runtime reconfigurable system by design, but also to express arbitrary contracts for each component which the overall system has to satisfy in order to run them. Additionally, we proposed a concept on how to enforce such contracts at runtime and therefore, easen the demonstration of semantic equivalence for newly added or exchanged components. Advantages explained in our former approaches still hold true, i.e., being able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices, as well as enhanced maintainability and better configurability.

Within the evaluation of this approach we have shown, that those additional benefits come with a cost, e.g., runtime contract enforcement comes with an additional computing cost resulting in 1/5 to 1/15 the throughput of the original system, depending on the type of contract, i.e., string based comparisons or merely the greater/smaller than check of an integer. Applied to the autopilot implementation of our former approaches, this means slower performance during runtime, but with a simplified provision of evidence in terms of contract compliance or semantic equivalence of newly added or exchanged components.

Part III.

FURTHER WORK AND CONCLUSIONS

6 Further Work

The first two approaches described in this work can be seen as a prototypical implementation of a model-driven, modular-by-design SCSS development methodology. The third approach complements this methodology by further strengthen module boundaries by declaring simple contracts between those modules, checking them at compile time and enforcing them at runtime.

Although those three approaches present a working prototype, they are probably not complete nor sufficient to support developers of SCSS during their everyday work. Therefore, in this chapter we present several different enhancements that in our opinion could be added to our approach in order to make it usable for real-world development scenarios.

The rest of this chapter is structured as follows: In section 6.1 we will show which additional parts of AADL should be added to the subset of source language constructs in order to enable the sufficient definition of a SCSS within AADL. In section 6.2 we will elaborate our thoughts on inter-process communication as this subject has been neglected in the aforementioned approaches. Finally, in section 6.3 we propose additional contract types that would be sensible, especially in a SCSS context and also elaborate on how to enhance developer experience of our contracts with additional tooling.

6.1. Extended Subset of AADL

In the aforementioned chapters we used merely a subset of AADL which we considered sufficient for our different approaches. This was done due to scope restrictions of this work, but also due to limitations posed by the chosen target language, i.e., RTSJ, and framework, i.e., OSGi. This subset however did neither capture the semantic possibilities given by AADL nor exhaust all capabilities of target language and framework. Therefore, we now give a glimpse of what other language features of AADL might be sensible to investigate further in order to leverage the most of AADL's semantic capabilities for SCSS.

The presented AADL subset in section 2.2 has been structured into three different sections, i.e., components, **`features`** and **`connections`** and properties. The proposed subset components encompassed **`packages`**, **`processes`**, **`threads`**, **`data`** and **`subprograms`**, where **`subprograms`** and **`data`** only played a minor role in our approaches so far. The first extension we would propose to our approach would therefore, be to encompass **`subprograms`** and **`data`** further into the existing approach.

Currently, **`subprograms`** are only used implicitly for the business logic executed during different lifecycle phases of a thread, e.g., `start`, `stop` or `dispatch`, where they also could be used to describe the structure of whole programming libraries, e.g., a `Math` library that can be used by business logic within the lifecycle phases of a thread, but whose code is not tied to one specific thread implementation. Listing 6.1 exemplarily depicts such a library described by the components **`subprogram group`**, **`subprogram`**, as well as by new **`features`** `in` and `out parameter` and `provides subprogram access`. The definition of such libraries would therefore, result in encompassing not only **`subprograms`** but also **`subprogram groups`** into the extended AADL subset as **`subprogram groups`** are used to group **`subprograms`** into sensible collections of functions that semantically belong together. `Subprogram groups` can also be used to group other **`subprogram groups`**, thus enabling authors of SCSS to structure their libraries hierarchically. Method parameters and return values can also be modeled by using **`in`** and **`out`** parameters, but might have to be restricted to only one **`out parameter`** per **`subprogram`** as many programming languages, including Java, only allow one return value per method. Another solution might incorporate a code generator that transforms **`subprograms`** defining more than one **`out parameter`** into Java methods with return parameters that have to be passed by reference. However this would then exclude primitive types, e.g., `Base_Types` in Listing 6.1, from being a valid data type for an **`out parameter`**.

```
subprogram group Math
features
    sub: provides subprogram access Sub;
    add: provides subprogram access Add;
end Math;

subprogram Sub
features
    input_one: in parameter Base_Types::float;
    input_two: in parameter Base_Types::float;
    result: out parameter Base_Types::float;
end Sub;

subprogram Add
features
    input_one: in parameter Base_Types::float;
    input_two: in parameter Base_Types::float;
    result: out parameter Base_Types::float;
end Add;
```

Listing 6.1: Definition of Math library via subprograms and subprogram group

Such trade-offs must be made in order to incorporate **`subprograms`** into our existing approaches.

The **`data`** component also would be a great addition to our approaches, as a developer of a SCSS then would be able to declare access and provisioning of shared data within other structures, e.g., **`subprograms`**, **`threads`** or **`processes`**. Currently, **`data`** is only used as type descriptor in the context of `data ports`, whereas **`data`** components in general can also be used to describe data structures that components can hold as global or local variable, e.g., to represent state or for sharing data with other components via shared data access. Such an access to a shared state variable is shown in Listing 6.2, where two **`subcomponents`** **`subcomp1`** and **`subcomp2`** access the same data **`sharedState`**, kept by their parent component **`process.impl`**. as can be seen in this example not only would an incorporation of data encompass processing additional properties like **`Concurrency_Control_Protocol`** or **`Access_Right`** but also additional **`features`**, i.e., **`data access`**. Similar to the aforementioned trade-offs, the incorporation of **`data`** components into the code generation would come with similar difficulties regarding their mapping onto target language and framework.

```
process implementation process.impl
subcomponents
    sharedState: data SharedState {Concurrency_Control_Protocol => Semaphore
        ;};
    subcomp1: thread Comp1;
    subcomp1: thread Comp2;
connections:
    access1: data access subcomp1 -> sharedState;
    access2: data access subcomp2 -> sharedState
end process.impl;

thread Comp1
features
    sharedState: requires data access SharedState {Access_Right => Write_Only
        ;};
end Comp1;

thread Comp2
features
    sharedState: requires data access SharedState {Access_Right => Read_Only
        ;};
end Comp2;
```

Listing 6.2: Definition of shared data in AADL

Aside from the aforementioned extensions to our approaches we would also recommend to include the following AADL language constructs into a further developed approach:

- **`Event Ports/Event Data Ports`**: Currently, our approach merely support synchronous communication through **`data ports`** and **`event ports`** are only used for triggering mode transitions. The inclusion of **`event ports`** and **`event data ports`** would extend our approach to be able to support asynchronous communication. Therefore, also constructs as message queues, queue overflows and similar mechanisms for event-based communication must be considered, as far as they are already dictated through AADL language constructs, e.g., through properties like **`Queue_Size`**, **`Overflow_Handling_Protocol`** or **`Queue_Processing_Protocol`**.

- **`Dispatch_Protocol`**: Currently, only **`periodic`** is supported for threads, due to the scope of this work. Other protocols should be part of an extended approach in order to support **`sporadic`**, **`aperiodic`** or **`background`** threads. Of course the inclusion of such protocols recursively implies the inclusion of other language constructs like **`event ports`** as for example **`sporadic threads`** are purely event triggered.

- Hardware components: Currently, we only support software related components like **`process`**, **`thread`** or **`data ports`**. In conjunction with our approach presented in chapter 5 it would be sensible to include hardware components too, e.g., **`processor`**, or **`memory`** which then can be used for the values of contract properties. Given the case a software component references a specific **`memory`** component, then the properties of this **`memory`** component can be used to determine the amount of memory needed by this software component and therefore, a contract could be generated, stating this amount of memory to be a mandatory requirement.

6.2. Inter-Process Communication

Another rather large enhancement would be the integration of inter-process communication into the existing approach. Until now we only have presented a transformation from AADL to RTSJ and OSGi that implies all software components to run in one process. Inter-process communication has been excluded due to the scope of this work, but not due to the capabilities offered by target language and framework. On the contrary, OSGi offers sufficient mechanisms that directly target the distribution of software components, i.e., bundles and components, over several processes, i.e., usually different machines or computing nodes. These mechanisms encompass OSGi's Remote Services [87] as well as the Remote Service Admin Service Specification [88]. Both can be found in OSGi's compendium specification and address the distribution of services over remote processes. Newer specifications even further enhance OSGi's distribution capabilities, namely the Cluster Information Specification [89] which can be utilized to monitor the resources of a remote process, e.g., CPU and memory consumption, and the REST Management Service Specification [90] which offers a standardized REST endpoint on each of the remote processes through which new bundles and services can be installed, started and stopped. Now, we first show how to export a formerly local-only service and then explain the concepts of the Remote Service Admin Service Specification which enables the export.

The export of a service in OSGi is fairly simple as can be seen in Listing 6.3. All a developer has to do is to add the property `service.exported.interfaces` to its component properties. The value of this property defines which of the component's services shall be exported and thus being able to be discovered by a remote process. In this case we decided to export all services, i.e., `interfaces`, by using the wildcard operator `*`.

```
@Component(
    property = {"service.exported.interfaces=*"}
)
public class ExportedService implements IExportedService { ... }
```

Listing 6.3: Definition of an exported service

The different components of an implementation of the Remote Service Admin Service Specification then take care of exporting the respective services and making it visible to other OSGi frameworks that have their own Remote Service Admin Service Specification implementation running. This way a developer of a component is freed completely from tasks like network communication, service discovery or endpoint registration. Those are handled by additional components like the Remote Service Admin, Topology Manager or a Discovery, whose interplay is depicted in Figure 6.1.

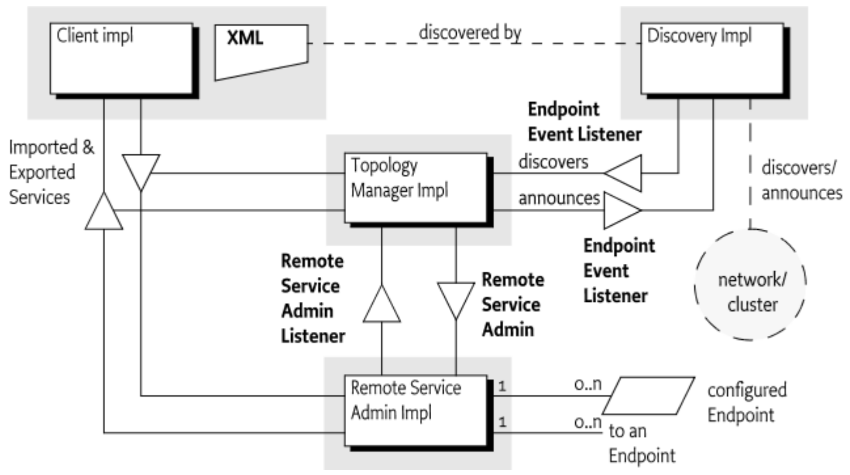

Figure 6.1.: Remote Service Admin Entities [88]

Topology Managers are responsible for the distribution policies of an OSGi framework, i.e., they listen to local services and decide which to expose. For each service that shall be exported a Topology Manager advises the Remote Service Admin to do the actual export. Topics like fail-over, monitoring or load-balancing can be addressed here transparently by using a suitable Remote Service Admin for these tasks.

A Remote Service Admin provides the basic mechanisms to import and export services, i.e., the `RemoteServiceAdmin interface` defines methods like `exportService()`, `importService()` and its counterparts `getExportedServices()` and `getImportedServices()`. Once advised by a Topology Manager to export a local service a Remote Service Admin will expose this service as a remote endpoint. Vice versa, when a Topology Manager advises the Remote Service Admin to import a Service it will create a local proxy for a remote endpoint.

Finally, a Discovery discovers remote endpoints and notifies local endpoint listeners about their presence as well as it publishes local endpoints for other OSGi frameworks.

Those three components and the mechanisms they provide for access to remote services can be used in a transformation from an AADL model that contains more than one **`process`** in order to enable the communication between **`thread`** components within them. Such a case is depicted in Figure 6.2 which is taken from subsubsection 2.2.3.3.

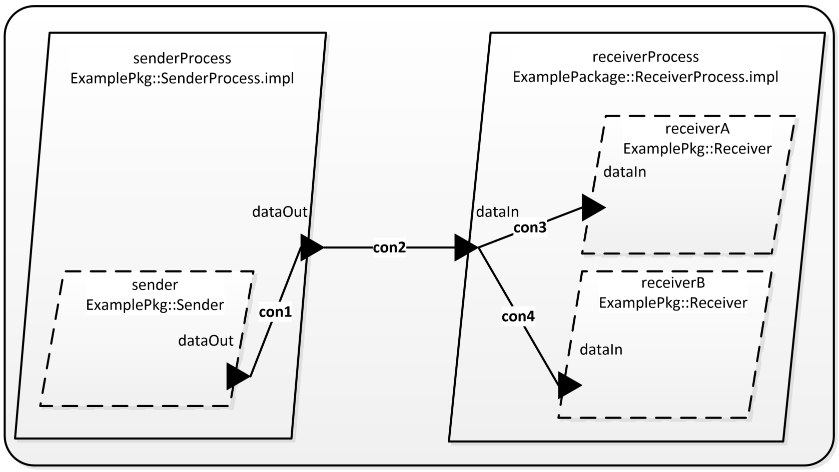

Figure 6.2.: Remote connection between two processes

Here, a `sender` thread within the **`senderProcess`** communicates with two receiver threads, i.e., **`threadA`** and **`threadB`**, that reside within another process, i.e., **`receiverProcess`**. Within Java those processes would usually be two different JVMs, each running in its own process, probably even on different physical machines. Therefore, a transformation from AADL to OSGi would have to take care of distributing the generated components and bundles accordingly. It must also ensure that the components can find each other and subsequently be able to communicate with each other in order to build a working system.

Currently, we translate each part of a semantic connection into an OSGi component, thus the transformation would need to decide within which process **`con2`** would have to reside, or in larger scenarios where not only one part of a semantic connection is outside of a process, how to aggregate those **`connections`** into one logical unit. As soon as a decision is made whether the part of a semantic connection that resides outside of any process is integrated into either sender or receiver, the transformation can then apply the aforementioned property `service.exported.interfaces` to this part of the semantic connection. Additionally, both processes have to incorporate implementations of the Remote Service Admin Service Specification, so that the remote service, i.e., a component representing **`con2`**, appears as a local one to the components residing within the other process.

This way, large systems spanning several processes could be defined within AADL and easily be transformed into a working OSGi system that is subsequently deployed to the respective physical machines.

However, this approach surely will pose several challenges in order to work dependably, e.g., additional metadata must be generated describing the deployments for each machine, encompassing not only the generated code, but also additional components necessary to form a working system, e.g., all components of an implementation for the Remote Service Admin Service Specification or additional components implementing specific communication protocols if needed. Also the logic deciding which parts of a semantic connection that are not defined within the boundaries of a process in AADL to deploy on which physical machines might pose challenges that are not yet obvious regarding edge-cases.

6.3. Enhanced Contracts

The last enhancements we propose affect the contracts presented in chapter 5. First, in subsection 5.3.1 we presented several possible contract types that we consider sensible in the context of SCSS. However, the presented ones only pose a fraction of what is possible or even sensible in the first place. Therefore, in order to enhance this approach even further we would propose several different topics that should be covered by future contract types. One good example for such additional contracts might be different Security Integrity Level (SIL) that can be attached to different components.

In this context we will use Automotive Security Integrity Level (ASIL) level, taken from ISO 26262, which can take a alphabetic letter value ranging from A to D, where D is the highest SIL and A the lowest one, respectively. Given the case we defined an additional contract for those, then a developer is able to force one component of a specific SIL to only be usable in conjunction with other components of the same or lower SIL. This way a developer can set a maximum SIL for the platform the components are running on, which can help during certification, as the platform usually has to certified for the highest SIL any of its component has.

Other additional contracts might be the extended hardware contract we used during evaluation, see section 5.5, where not only CPU and RAM are given by the contract, but also a generic description of sensors or actuators the component needs to be runable.

Another rather large enhancement would be the addition of a generator for those contracts right from the AADL model. Some of the information needed for the contracts proposed in this work are already contained within the AADL models, e.g., used hardware and also type value ranges would be possible. Others would need the definition of an additional property set in AADL. Property sets are an extension mechanism provided by AADL so a developer can define his own properties for specific AADL component types, e.g., a property of type `String` called pre-condition for all components of type **`subprogram`**, or its counterpart a post-condition.

Those additional properties then could be used during transformation to automatically generate requirements and capabilities or component property types accordingly for the different components. Thus, freeing the developer from writing those capabilities/requirements/component property types by himself.

7 Conclusion and Outlook

This work presented three stacking approaches.

First, we presented an approach that offers a transformation between AADL and plain RTSJ, while keeping the semantics defined in the AADL model untouched. This approached aimed at proving the general feasibility of such a transformation in order to achieve a more abstract goal defined in subsection 1.2.1, i.e., providing a methodology for SCSS, which enables developers to reduce common errors and enables them to discover uncommon errors during early stages of the development process.

The solution proposed in chapter 3 therefore, offers a possible solution to tackle the accompanying problems also defined in subsection 1.2.1, i.e., increasing cost of maintenance and operations in relation to cost for actual development, time wasted for discovery of failures found during testing instead of during design phases and delayed delivery resulting from the two aforementioned problems. Our approach enables developers of SCSS to shift structure, timing and communication-related concerns into design phase. Hence, they are able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices. The application of our approach is shown via the implementation of an autopilot for quadrocopter and shows three advantages of this approach over an implementation without code-generation, i.e., the speed-up of development by letting the programmer focus on application logic, a less error-prone transition from the design of a system to its implementation and the possibility of an earlier detection of timing- or communication-related errors in the system.

Second, we tackled shortcomings of the first approach regarding maintainability of the generated code base while at the same time tackling the challenge defined in subsection 1.2.2, i.e., providing a possibility for developers of SCSS, to enhance maintainability of the final system by design and enabling hot updates of running systems during design time. In order to overcome this challenge we introduced an additional layer of indirection to our technology stack by introducing OSGi as an underlying runtime framework but also as a methodology for common issues regarding modularity of Java code, like encapsulation on .jar level, dependency injection or a matured dependency model between .jars. By using OSGi as an underlying modularity framework we had to change some basic transformation rules formerly established by our first approach in favor of a more modular generated code base, i.e., how inheritance is handled or how `features` are represented within the generated code. This approach enables developers of a SCSS to not only shift structure, timing and communication-related concerns into design phase, but also enables them to create a highly modular, runtime reconfigurable and, most important, a more easy to maintain system by design. Advantages explained in our former approach still hold true, i.e., being able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices, whereby new advantages, i.e., enhanced maintainability and better configurability are added on top. Problems as stated in subsection 1.2.2, i.e., updates being not allowed to restart the system or not being allowed to affect other parts of the system, as well as systems not being easily extendable, could be solved as was shown during our evaluation. However, during the same evaluation we also have shown that those additional benefits come with a cost, e.g., configurability comes with an additional computing cost as well as an increased memory consumption. Therefore, the benefits provided by our approach always have to be evaluated against its drawbacks within the context of a specific system design.

Third, we enhanced the semantic expressiveness of pure `interfaces` of OSGi services by providing different requirement-capability definitions that can be leveraged during runtime by an OSGi resolver to determine if a new service implementation is semantically equivalent to an old one. The proposed requirement-capability definitions encompass pre-/post-conditions as well as WCET statements and also hardware dependencies, so that not only dependencies between software modules are expressible, but also dependencies between software modules and the underlying hardware that is a mandatory requirement to run these software modules. Finally, we introduced a concept allowing us to enforce the contracts of modules at runtime and centralizing contract violation handling independently from the components defining the respective contract. This approach tackled the challenge defined in subsection 1.2.2, i.e., enable developers of SCSS to easily show semantic equivalence of their updates in order to easen the certification of partial updates of existing systems. The approach shown in chapter 5 thus provides developers with a possibility to narrow down the semantic borders of a component so far that a partial certification is easend in terms of demonstrating that one component can only be exchanged with another if it is semantically equivalent to another one. The feasibility of our approach was shown through a set of qualitative, scenario-based evaluation cases. Additionally, a qualitative evaluation was done which examined the impact of runtime contract enforcement on message passing in comparison to a solution without contracts. This evaluation indicated roughly a 80% performance loss in comparison to a non-contract solution.

The presented approaches form an appropriate foundation for the aforementioned enhancements of model-driven development of highly modular SCSS and thus, serve as a suitable basis for further research.

Part IV.

ANNEX

List of Abbreviations

AADL	Architecture Analysis and Design Language
RTSJ	Real-Time Specification for Java
OSGi	Open Services Gateway initiative
JSE	Java Standard Edition
MDD	Model-Driven Development
JVM	Java Virtual Machine
.jar	Java Archive
SLOC	Source Lines of Code
USAF	United States Air Force
JIT	Just In Time
AOT	Ahead Of Time
JSR	Java Specification Request
LED	Light Emitting Diod
JNI	Java Native Interface
API	Application Programming Interface
JRE	Java Runtime Environment
DS	Declarative Services
CDI	Contexts and Dependency Injection

DI	Dependency Injection
OS	Operating System
DS	Declarative Service
LDAP	Lightweight Directory Access Protocol
FODA	Feature-Oriented Domain Analysis
UML	Unified Modeling Language
LoC	Lines of Code
UML	Unified Modeling Language
REST	Representational State Transfer
SCSS	Safety-Critical Software Systems
CGSS	Consumer-Grade Software Systems
SysML	Systems Modeling Language
CB	Connection Broker
SCR	Service Component Runtime
JMH	Java Microbenchmark Harness
BPEL	Business Process Execution Language
WSDL	Web Service Description Language
JDL	JHipster Domain Language
IDE	Integrated Development Environment
PJCD	Portable Java Contract Definitions

WCET	Worst-Case Execution Time
DSL	Domain-Specific Language
JMH	Java Microbenchmark Harness
SIL	Security Integrity Level
ASIL	Automotive Security Integrity Level
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language

Bibliography

[1] Christian Hagen and Jeff Sorenson. "Delivering military software affordably". In: *Defense AT&L* (2013), pp. 30–34.

[2] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

[3] Gary McGraw. *Software security: building security in*. Vol. 1. Addison-Wesley Professional, 2006.

[4] StackOverflow. *Stack Overflow Trends*. URL: `https://insights.stackoverflow.com/trends?tags=java%2Cc%2Cassembly` (visited on 01/29/2019).

[5] Appian. *Appian Platform*. URL: `https://de.appian.com/platform/` (visited on 01/29/2019).

[6] Google. *Google Trends*. URL: `https://trends.google.de/trends/explore?date=all&q=model%20driven%20development,low%20code%20platform` (visited on 01/29/2019).

[7] Strategic Planning. "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology* (2002).

[8] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.

[9] David A Wheeler. "Ada, C, C++, and Java vs. the Steelman". In: *ACM SIGAda Ada Letters* 17.4 (1997), pp. 88–112.

[10] Tiobe software BV. *Tiobe Index*. Oct. 1, 2017. URL: `https://www.tiobe.com/tiobe-index/` (visited on 01/29/2019).

[11] Thomas Driessen and Bernhard Bauer. "Shifting temporal and communicational aspects into design phase via AADL and RTSJ". In: *Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*. IEEE. 2016, pp. 1–10.

[12] Thomas Driessen et al. "Layered-V". In: *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*. IEEE. 2015, 10A2–1.

[13] Laminar Research. *XPlane 11*. URL: `https://www.x-plane.com/` (visited on 01/29/2019).

[14] Erle Robotics. *Erle Copter Drone Kit*. URL: `https://erlerobotics.com/blog/product/erle-copter-diy-kit/` (visited on 01/29/2019).

[15] aicas GmbH. *JamaicaVM*. URL: `https://www.aicas.com/cms/en/JamaicaVM` (visited on 01/29/2019).

[16] OpenJDK. *Code Tools: jmh*. URL: `https://openjdk.java.net/projects/code-tools/jmh/` (visited on 01/29/2019).

[17] Julien Ponge. *Avoiding Benchmarking Pitfalls on the JVM*. URL: `https://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html` (visited on 01/29/2019).

[18] Carnegie Mellon Software Engineering Institute. *Architecture Analysis and Design Language*. URL: `http://www.aadl.info/aadl/currentsite/` (visited on 01/29/2019).

[19] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[20] James J Hunt. *Realtime and Embedded Specification for Java - Version 2.0*. Jan. 8, 2019. URL: `https://www.aicas.com/cms/sites/default/files/rtsj_68.pdf` (visited on 01/29/2019).

[21] *The Real-Time Specification for Java 2.0*. URL: `https://www.aicas.com/cms/en/rtsj` (visited on 01/29/2019).

[22] Andrew J Wellings. *Concurrent and real-time programming in Java*. John Wiley, 2004.

[23] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, 2009.

[24] Eric J Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Pearson Education, 2009.

[25] The Linux Foundation. *Cyclictest*. June 22, 2017. URL: `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start` (visited on 01/29/2019).

[26] Emlid. *Raspberry Pi Real-Time Kernel*. May 19, 2014. URL: `https://emlid.com/raspberry-pi-real-time-kernel/` (visited on 01/29/2019).

[27] Bill Joy et al. *The Java language specification*. 2000.

[28] aicas GmbH. *JamaicaVM 8.0 - User Manual*. Mar. 9, 2016. URL: `https://www.aicas.com/cms/sites/default/files/jamaicavm_8.0_manual.pdf` (visited on 01/29/2019).

[29] aicas GmbH. *RTSJ 2.0 JavaDoc*. URL: `https://www.aicas.com/rtsj/Version_2_0/all/` (visited on 01/29/2019).

[30] OSGi Alliance. *Architecture*. URL: https://www.osgi.org/developer/architecture/ (visited on 01/29/2019).

[31] The Apache Foundation. *Apache Celix*. URL: https://celix.apache.org/ (visited on 01/29/2019).

[32] OSGi Alliance. *Benefits of Using OSGi*. URL: https://www.osgi.org/developer/benefits-of-using-osgi/ (visited on 01/29/2019).

[33] Heiko Seeberger. *OSGi in kleinen Dosen – Bundles und Life Cycle*. Feb. 2, 2009. URL: https://jaxenter.de/osgi-in-kleinen-dosen-bundles-und-life-cycle-2-8999 (visited on 01/29/2019).

[34] OSGi Alliance. *org.osgi.service.component.annotation*. URL: https://osgi.org/javadoc/osgi.cmpn/7.0.0/org/osgi/service/component/annotations/package-frame.html (visited on 01/29/2019).

[35] OSGi Alliance. *Declarative Services Specification*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/service.component.html (visited on 01/29/2019).

[36] Tim Howes. *A String Representation of LDAP Search Filters*. URL: https://www.ietf.org/rfc/rfc1960.txt (visited on 01/29/2019).

[37] Dirk Fauth. *Getting Started with OSGi Declarative Services*. June 21, 2016. URL: http://blog.vogella.com/2016/06/21/getting-started-with-osgi-declarative-services/ (visited on 01/29/2019).

[38] Oracle Corporation. *JSR 299: Contexts and Dependency Injection for the Java EE platform*. URL: https://www.jcp.org/en/jsr/detail?id=299 (visited on 01/29/2019).

[39] Google Inc. *Guice*. URL: https://github.com/google/guice (visited on 01/29/2019).

[40] Martin Fowler. "Inversion of control containers and the dependency injection pattern". In: (2004).

[41] Peter Kriens and B Hargrave. "Listeners considered harmful: The 'whiteboard' pattern". In: *Technical whitepaper, OSGi Alliance* (2004).

[42] OSGi Alliance. *Patterns*. URL: https://enroute.osgi.org/FAQ/400-patterns.html (visited on 01/29/2019).

[43] OSGi Alliance. *OSGi Core Release 7*. URL: https://osgi.org/specification/osgi.core/7.0.0/index.html (visited on 01/29/2019).

[44] OSGi Alliance. *OSGi Compendium Release 7*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/index.html (visited on 01/29/2019).

[45] Oracle. *Default Methods*. URL: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html (visited on 01/29/2019).

[46] Matteo Bordin et al. "System to Software Integrity: A Case Study". In: *Embedded Real-Time Software and Systems 2014.* , FR, 2014. URL: http://oatao.univ-toulouse.fr/10939/.

[47] Jérôme Hugues. *Open AADL*. URL: http://www.openaadl.org/ocarina.html (visited on 01/29/2019).

[48] Mathworks. *Simulink: Simulation und Model-Based-Design*. URL: http://de.mathworks.com/products/simulink/ (visited on 01/29/2019).

[49] Ying Wang et al. "Automatic RT-Java code generation from AADL models for ARINC653-based avionics software". In: *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE. 2012, pp. 670–679.

[50] Sam Procter and John Hatcliff. "Robby: towards an AADL-based definition of app architectures for medical application platforms". In: *Proceedings of the International Workshop on Software Engineering in Healthcare. Washington, DC*. 2014.

[51] Bodeveix Jean-Paul et al. "A mapping from AADL to Java-RTSJ". In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. ACM. 2007, pp. 165–174.

[52] Philip Mayer, Andreas Schroeder, and Nora Koch. "MDD4SOA: Model-driven service orchestration". In: *12th International IEEE Enterprise Distributed Object Computing Conference*. IEEE. 2008, pp. 203–212.

[53] Martin Wirsing et al. "Semantic-based development of service-oriented systems". In: *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer. 2006, pp. 24–45.

[54] Simon Johnston. "UML 2.0 profile for software services". In: *IBM developerWorks http://www. ibm. com/developerworks/rational/library/05/419_soa* (2005).

[55] David Skogan, Roy Grønmo, and Ida Solheim. "Web service composition in UML". In: *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*. IEEE. 2004, pp. 47–57.

[56] Luciano Baresi, Reiko Heckel, Sebastian Thöne, et al. "Style-based modeling and refinement of service-oriented architectures". In: *Software & Systems Modeling* 5.2 (2006), pp. 187–207.

[57] Floriment Klinaku and Vincenzo Ferme. “Towards Generating Elastic Microservices: A Declarative Specification for Consistent Elasticity Configurations”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2018, pp. 510–513.

[58] Matt Raible. *The JHipster mini-book*. Lulu.com, 2016.

[59] Philip Wizenty et al. “Magma: Build management-based generation of microservice infrastructures”. In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ACM. 2017, pp. 61–65.

[60] Apache. *Welcome to Apache Maven*. URL: `https://maven.apache.org/` (visited on 01/29/2019).

[61] Nataliya Yakymets et al. “Model-driven safety assessment of robotic systems”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2013, pp. 1137–1142.

[62] Miroslav Pajic et al. “Model-driven safety analysis of closed-loop medical systems”. In: *IEEE Transactions on Industrial Informatics* 10.1 (2014), pp. 3–16.

[63] Lars Grunske, Bernhard Kaiser, and Yiannis Papadopoulos. “Model-driven safety evaluation with state-event-based component failure annotations”. In: *International Symposium on Component-Based Software Engineering*. Springer. 2005, pp. 33–48.

[64] Angelo Gargantini and Constance Heitmeyer. “Using model checking to generate tests from requirements specifications”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 24. 6. Springer-Verlag. 1999, pp. 146–162.

[65] Mehdi Malekzadeh and Raja Noor Ainon. “An automatic test case generator for testing safety-critical software systems”. In: *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*. Vol. 1. IEEE. 2010, pp. 163–167.

[66] Sven Burmester et al. “The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems”. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 670–671.

[67] Tommaso Cucinotta et al. “A real-time service-oriented architecture for industrial automation”. In: *IEEE Transactions on industrial informatics* 5.3 (2009), pp. 267–277.

[68] Akihito Iwai and Mikio Aoyama. “Automotive cloud service systems based on service-oriented architecture and its evaluation”. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 638–645.

[69] OSGi Alliance. *Configurator Specification*. URL: `https://osgi.org/specification/osgi.cmpn/7.0.0/service.configurator.html` (visited on 01/29/2019).

[70] Antoine Beugnard et al. "Making components contract aware". In: *Computer* 32.7 (1999), pp. 38–45.

[71] Bertrand Meyer. "Applying'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[72] Albert Benveniste et al. "Contracts for systems design: theory". PhD thesis. Inria Rennes Bretagne Atlantique; INRIA, 2015.

[73] OSGi Alliance. *Dependencies*. URL: `https://osgi.org/specification/osgi.core/7.0.0/framework.module.html#framework.module.dependencies` (visited on 01/29/2019).

[74] OSGi Alliance. *Framework Namespaces Specification*. URL: `https://osgi.org/specification/osgi.core/7.0.0/framework.namespaces.html` (visited on 01/29/2019).

[75] OSGi Alliance. *Semantic versioning–technical whitepaper*. Tech. rep. Technical report, OSGi Alliance, 2010.

[76] OSGi Alliance. *Filter Syntax*. URL: `https://osgi.org/specification/osgi.core/7.0.0/framework.module.html#framework.module.filtersyntax` (visited on 01/29/2019).

[77] OSGi Alliance. *Service Hook Service Specification*. URL: `https://osgi.org/specification/osgi.core/7.0.0/framework.servicehooks.html` (visited on 01/29/2019).

[78] Oracle. *Dynamic Proxy Classes*. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html` (visited on 01/29/2019).

[79] Mike Barnett and Wolfram Schulte. "Contracts, components, and their runtime verification on the .NET platform". In: *J. Systems and Software, Special Issue on Component-Based Software Engineering* (2002).

[80] Egon Börger. *Specification and validation methods*. Clarendon Press, 1995, pp. 9–36.

[81] Reto Kramer. "iContract-the Java/sup TM/design by Contract/sup TM/-tool". In: *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*. IEEE. 1998, pp. 295–307.

[82] Dean Wampler. "Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces". In: *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. 2006, pp. 27–30.

[83] Joel Costigliola. *AssertJ*. URL: https://github.com/joel-costigliola/assertj-core (visited on 01/29/2019).

[84] Pact Foundation. *Pact - Introduction*. URL: https://docs.pact.io/ (visited on 01/29/2019).

[85] OSGi Alliance. *Portable Java Contract Definitions*. URL: https://www.osgi.org/portable-java-contract-definitions/ (visited on 01/29/2019).

[86] NASA JPL. *Mars Climate Orbiter Mishap Investigation Board Phase I Report*. URL: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf (visited on 01/29/2019).

[87] OSGi Alliance. *Remote Services*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/service.remoteservices.html (visited on 01/29/2019).

[88] OSGi Alliance. *Remote Service Admin Service Specification*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/service.remoteserviceadmin.html (visited on 01/29/2019).

[89] OSGi Alliance. *Cluster Information Specification*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/service.clusterinfo.html (visited on 01/29/2019).

[90] OSGi Alliance. *REST Management Service Specification*. URL: https://osgi.org/specification/osgi.cmpn/7.0.0/service.rest.html (visited on 01/29/2019).

List of Figures

List of Tables