TU WIEN Informatics

AADL Modeling for OPC UA/DDS Gateway

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Daniel Scheuchenstuhl
Matrikelnummer 01630368

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Patrick Heinrich Denzler
Dipl.-Ing. Daniel Ramsauer

Wien, 17. September 2020

Daniel Scheuchenstuhl   Wolfgang Kastner

Informatics

AADL Modeling for OPC UA/DDS Gateway

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Daniel Scheuchenstuhl
Registration Number 01630368

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Wolfgang Kastner
Assistance: Dipl.-Ing. Patrick Heinrich Denzler
Dipl.-Ing. Daniel Ramsauer

Vienna, 17th September, 2020

Daniel Scheuchenstuhl	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Daniel Scheuchenstuhl

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. September 2020

Daniel Scheuchenstuhl

Danksagung

Zu aller erst möchte ich meinen Betreuern Dipl.-Ing. Patrick Denzler und Dipl.-Ing. Daniel Ramsauer für die kontinuierliche und umfassende Unterstützung im Rahmen dieser Arbeit danken. Besonders möchte ich dabei Dipl.-Ing. Patrick Denzler für sein außerordentliches Engagement honorieren.

Darüber hinaus möchte ich mich an dieser Stelle bei meinen Eltern, meinem Onkel und meinen Großeltern für die tatkräftige, motivierende und finanzielle Unterstützung im Rahmen des gesamten Studiums bedanken, ohne die ein derartig schneller Fortschritt in meinem Studium nicht möglich gewesen wäre.

Acknowledgements

First of all, I would like to thank my assistant supervisors, Dipl.-Ing. Patrick Denzler and Dipl.-Ing. Daniel Ramsauer for their continuous and comprehensive support in the context of this work. I would particularly like to honor Dipl.-Ing. Patrick Heinrich Denzler for his extraordinary commitment.

In addition, I would like to take this opportunity to thank my parents, my uncle and my grandparents for their energetic, motivating and financial support throughout my study, without which such rapid progress in my study would not have been possible.

Kurzfassung

Fabriken der nächsten Generation implementieren das Konzept des Industrial Internet of Things (IIoT) zur Integration hochentwickelter, interoperabler, skalierbarer und effizienter Automatisierungssysteme. Um eine robuste und zuverlässige Kommunikation zu gewährleisten, wurden die Kommunikationsstandards Open Portable Communication Unified Architecture (OPC UA) und Data Distribution Service (DDS) etabliert, welche jedoch unterschiedliche Kommunikationsschemata verwenden. Da beide Standards in entgegengesetzten Anwendungsdomänen verwendet werden, implementieren verschiedene Geräte entsprechend deren Anwendungskontext entweder den einen oder den anderen Standard. Um die notwendigen Anforderungen in modernen Automatisierungssystemen zu erfüllen, ist insbesondere die Interoperabilität der integrierten Geräte von großer Bedeutung. Zu diesem Zweck zielt ein vollständig konfigurierbares formales Modell des OPC UA/DDS Gateways darauf ab, die IIoT Anforderungen zu gewährleisten. Um dieses Ziel zu erreichen, präsentiert diese Arbeit den gesamten Implementierungsprozess des OPC UA/DDS Gateway AADL Modells, die Konformitätsbewertung sowie die spezifischen Lernergebnisse. Darüber hinaus wird in dieser Arbeit ein Verfahren vorgestellt, anhand dessen das formale allgemeine AADL Modell konfiguriert werden kann, um eine ausführbare spezifische Gateway-Instanz zu erhalten. In Bezug auf die angegebenen Abgrenzungen wird das AADL Modell anstelle einer ausführbaren Binärdatei zur Weiterentwicklung bereitgestellt. Zukünftige Forschung sollte sich auf die aufgetretenen Probleme bei der Codegenerierung, die Modellierung bestimmter OPC UA und DDS Entitäten in AADL und die Erweiterung des allgemeinen Modells konzentrieren.

Abstract

Next-generation factories implement the concept of the Industrial Internet of Things (IIoT) for integrating sophisticated, interoperable, scalable and efficient automation systems. On the level of communication, Open Portable Communication Unified Architecture (OPC UA) and Data Distribution Service (DDS) mature to possible standards for robust and reliable automation systems protocols. The possibility to use both protocols in various application domains and devices makes them valid candidates. However, OPC UA and DDS follow two different communication schemes, (i.e. server/client and publish/subscribe) what makes communication between the protocols very difficult. For still being able to achieve high interoperability as envisioned in IIoT, a possible solution is a complete configurable OPC UA/DDS Gateway. This thesis supports the creation of such a gateway by illustrating the entire implementation process of an OPC UA/DDS Gateway AADL model, the conformance evaluation, as well as the specific learning outcomes. Moreover, this thesis introduces a procedure on how to configure the formal general AADL model in order to acquire an executable specific gateway instance. The created AADL model can be used for further development. Future research should focus on the encountered code generation issues, how to model particular OPC UA and DDS entities in AADL and the extension of the general model.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Scientific Background **5**
2.1 Related Work 5
2.2 OPC UA 7
2.3 DDS 9
2.4 OPC UA/DDS Gateway 11
2.5 Information Models 14
2.6 AADL Modeling 15

3 Scientific Methodology **27**
3.1 OSATE 28
3.2 Ocarina 28
3.3 Chronological Development Process 29

4 AADL Model for the OPC UA/DDS Gateway **31**
4.1 General Structure of the AADL system model 31
4.2 AADL Model for the OPC UA to DDS Bridge 32
4.3 AADL Model for the DDS to OPC UA Bridge 33
4.4 Modeling of OPC UA Components 33
4.5 Modeling of DDS Components 39

5 Dynamic Configurability of the OPC UA/DDS Gateway **49**
5.1 Code generation process using Ocarina 49
5.2 Experienced Problems 51
5.3 A dynamically configurable OPC UA/DDS Gateway application . . . 53

6 Discussion on the dynamically configurable OPC UA/DDS Gateway **57**

6.1 Conformance of the OPC UA/DDS Gateway AADL model with the OPC UA/DDS Gateway specification 57
6.2 Discussion of the results regarding the research questions 59

7 Conclusion 61

8 Appendix 63
8.1 AADL Code for the OPC UA/DDS Gateway system model 63
8.2 Safety Analysis in AADL 63
8.3 Analysis Methods 68

List of Figures 71

Acronyms 73

Bibliography 75

CHAPTER 1

Introduction

As the production throughput is supposed to steadily increase and the production methods become more and more complex, modern factories have to integrate highly sophisticated, scalable and efficient automation systems to meet all necessary requirements [1]. Most modern automation systems attempt to implement the concept of the Industry 4.0 standard or also known as the Industrial Internet of Things (IIoT) [2]. IIoT intends on integrating a vast number of low-level field devices into a common distributed cloud system so that communication, monitoring and maintenance over all layers of the automation pyramid is vastly improved [3]. Especially at the Programmable Logic Controller (PLC) level and the sensor/actuator level of the automation pyramid [4], up to millions of different devices are in use which communicate with each other and are supposed to exchange data while there may not be any guarantees about the interoperability of the individual devices. Due to the wide number of manufacturers, many devices support different communication protocols and communication schemes which are designed to efficiently perform in some specific application context [5]. Although standards in terms of low-level communication protocols like ASI and Profibus have been introduced and are comprehensively used, major interoperability issues between the protocols integrated in the lower levels are still present [3]. For this reason, the Open Portable Communication Unified Architecture (OPC UA) [6] and the Data Distribution Service (DDS) [5] standards were introduced at the Supervisory Control and Data Acquisition (SCADA) [7] level to resolve this issues. OPC UA and DDS are designed to provide interoperability, flexibility, scalability and efficient and robust data communication. Yet, they implement completely different approaches in terms of overall system and communication design which makes it impossible to combine the benefits of both system designs without implementing any kind of gateway [8]. For this purpose, the Object Management Group, Inc. (OMG) introduced the specification for an OPC UA/DDS Gateway solution which is intended to represent a standard for bidirectional communication between OPC UA and DDS [8].

Supplementary, the goal of the OPC UA/DDS Gateway specification is to establish well-defined conformance points. These allow specific OPC UA/DDS Gateway implementations to either provide basic or complete communication support between OPC UA and DDS entities depending on the implemented OPC UA to DDS Mapping and the implemented DDS to OPC UA Mapping. The specification comprehensively describes and defines the mapping of OPC UA entities to the corresponding DDS entities and vice versa, yet, few implementations exist while some only partially implement the standard and consider specific cases relevant for the individual application context [9]. Moreover, there are no formal model implementations, yet. An complete formal model implementation of the OPC UA/DDS Gateway would have the benefit of allowing to perform a full analysis of the model and later on to build a more optimized implementation or even generate code for different platforms and languages based on the formal model. Furthermore, an implementation generated from a formal model has the advantage that the implementation fully complies with the standard provided by the specification as the formal model already complies with the specification. The implementation may also be generated for multiple platforms and languages individually without performing major changes on the formal model. In order to be able to formally describe the software and the hardware specifications of the target platform necessary for the OPC UA/DDS Gateway model, a formal description language unlike the Unified Modeling Language (UML) which does not only allow the modeling of software components of a system is required. An extended formal description language is needed which also allows to specify real-time constraints and the hardware properties for the target architecture. For this reason, the Architecture Analysis & Design Language (AADL) [10] is introduced at this point.

AADL is a formal architecture description language which allows to model the software components as well as the hardware components of a system. The software components are generally represented by processes, threads, subprograms or data models. The hardware components are generally represented by processors, memories, buses and devices. AADL further supports the specification of properties for each component type. In general, the design of real-time, reliable and safety-critical systems proves to be a challenging and error-prone task. The formal verification of AADL system models as stated in [11] and [12] illustrates possible design flaws e.g. in terms of real-time constraints or safety requirements and ensures the development of a refined system implementation by supporting the overall development process of real-time, reliable and/or safety-critical systems.

Moreover, few implementations as provided by [9] and [13] attempt on connecting OPC UA and DDS. Unfortunately, those implementations are not based on a formal model and simply rely on a static mapping of OPC UA entities to DDS entities and vice versa. Based on this approach, the implementation may be optimized for some specific application context, but must be redesigned or reprogrammed if the application context is intended to be changed e.g. a new OPC UA type or DDS type is added to the model. This does not only result in a lot of engineering overhead, but also increases the cost and the time required for comitting these changes. In order to create a flexible and scalable system implementation, another approach for the design of a dynamically configurable system implementation has to be applied.

To that end, this thesis discusses and implements the development of an OPC UA/DDS Gateway AADL system model that allows the generation and configuration of platform-specific implementations, instances and code. The basic principle is described in two parts. In a first step, the OPC UA/DDS Gateway is modeled in OSATE [14] using AADL according to the specification provided by the OMG [8]. Later on, the platform-specific OPC UA/DDS Gateway implementation is generated from the formal OPC UA/DDS Gateway AADL system model. For this purpose, the Ocarina toolchain [15] is used which integrates the Polyorb Hi-C middleware in order to generate a C code implementation for a Linux/Unix 64bit system architecture based on the formal OPC UA/DDS Gateway AADL specification.

Therefore the aim of this thesis is to model the OPC UA/DDS Gateway using AADL and to discuss on how the OPC UA/DDS Gateway can be dynamically configured based on the formal AADL model. In order to achieve this research aim, the following research questions have been formulated:

RQ 1: How to implement the OPC UA/DDS Gateway in AADL?

RQ 2: How to dynamically configure the OPC UA/DDS Gateway for several application platforms?

The focus of the work relies on the modeling of the OPC UA/DDS Gateway in AADL and the discussion on how the OPC UA/DDS Gateway can be configured based on the formal AADL model. Thus, the following delimitations are made. This thesis does not implement the verification of the OPC UA/DDS Gateway AADL system model in terms of several safety analysis methods. Furthermore, this thesis does not provide a concrete implementation or a binary executable of the OPC UA/DDS Gateway.

Another objective of this thesis is to provide an Open Source AADL code for the OPC UA/DDS Gateway model implementation.

CHAPTER 2

Scientific Background

According to the state-of-the-art on this research topic and the stated problem description in *Chapter 1*, this chapter intends on giving a basic overview of the scientific work that has already been done on developing approaches to establish a communication between OPC UA and DDS entities. Furthermore, relevant scientific work that has been done on similar research topics as well as the technical background this thesis builds on is discussed. For this reason, the first section describes the related work while the second section and onward provide a principle introduction on the technical background of this thesis.

2.1 Related Work

Preceding this thesis, few scientific research has been done on this topic. The following paragraphs summarize the approaches and main points of the related work and outline the foundation of this thesis.

The design of real-time systems emerges to be a vastly error-prone development process as the analysis of real-time systems is a challenging task which arises the need for a comprehensive formal verification model. For this purpose, Hana Mkaouar, Bechir Zalila, Jérôme Hugues and Mohamed Jmaiel illustrate the formal verification of AADL real-time system models by defining formal semantics of an AADL behavioural subset using the LNT language [11]. Furthermore, they demonstrate their results in form of a robot case study.

But as challenging as the analysis of real-time systems, the safety-critical software engineering process also represents a delicate task. Thus, Hana Mkaouar, Bechir Zalila, Jérôme Hugues and Mohamed Jmaiel state an approach to formally map a real-time task model and introduce a formal verification phase in terms of an AADL model-based software engineering development process [12]. In addition to their results shown with the Flight control system and Line follower robot case studies, they provide a tool-chain for an automatic model transformation and formal verification of AADL models.

Aside from the formal verification of real-time systems modeled in AADL, few results have also been achieved to lay the foundations for a formal AADL system model of the OPC UA/DDS Gateway specification.

As OPC UA represents one of the main industrial standards and messaging protocols for the client-server communication model [6] and DDS represents one of the main industrial standards and messaging protocols for the publish-subscribe communication model [5], the idea to combine the benefits of these two system architectures arises. For achieving this objective, Ranti Endeley, Tom Fleming, Nanlin Jin, Gerhard Fehringer and Steve Cammish proposed the concept of a smart gateway in form of a middleware connecting the OPC UA architecture and the DDS system [9]. In their article, they illustrate how their middleware solution works and evaluate their concept by an application scenario using a Raspberry Pi.

The approach proposed in [13] tackles the idea to combine the benefits of both, OPC UA features and DDS features, in terms of a hybrid implementation. In their article, Julius Pfrommer, Sten Grüner and Florian Palm define the mapping of OPC UA data types to DDS data types and give a set of DDS QoS policies that fulfill the requirements of the OPC UA QoS specification.

The latter two concepts challenge the goal of an interoperable and comprehensive software solution to efficiently interconnect the OPC UA standard and the DDS standard in terms of the IIoT. But as these approaches set the basics for a realisation, they do not provide a mature formal model of the specification. A formal system model integrating the software and the hardware architecture would allow to refine a possible system implementation and could be used for a more specific and extensive system analysis concerning several system issues. For that reason, this thesis builds upon the insights of the related work and states a formal AADL system model of the OPC UA/DDS Gateway specification with the primary focus on the configuration aspect, respectively.

2.2 OPC UA

The abbreviation OPC UA [16] is an acronym for *Open Portable Communication Unified Architecture* and represents an industrial standard introduced with the IEC 62541 norm. The standard specifies a generic infrastructure model according to the concept of the automation pyramid with the purpose of establishing an efficient, robust, scalable and interoperable horizontal and vertical data exchange. Fundamental components of the OPC UA system architecture are the information and communication model as well as the message passing and the compliance model. These models define the structure of the passed information, the behaviour of the available services, the general information and control flow and the semantics of the information and how they have to be interpreted by high-level applications and field devices alike.

The OPC UA system architecture is based on an object-oriented modeling approach and relies on the classical client-server communication model. OPC UA clients are composed of an OPC UA communication stack, an integrated API and the client application which is shown in Figure 2.1. OPC UA servers are composed of an OPC UA communication stack, an integrated API and the server application which is shown in Figure 2.2. OPC UA servers define specific sets of services to interact with the resources of a server. These services are accessible by any OPC UA clients allowing them to connect to the address space of the appropriate OPC UA server. The address space of an OPC UA server is represented by a coherent graph of nodes which may be ordered hierarchically but may also have references to different nodes of any level. Each node identifies an object which relates to a resource of the server and is characterized by a type, attribute values, object relationships, methods and events. The topology of the graph may depend on the object relationships but generally the organisation of the address space only depends on the needs of the server application and can be adapted dynamically.

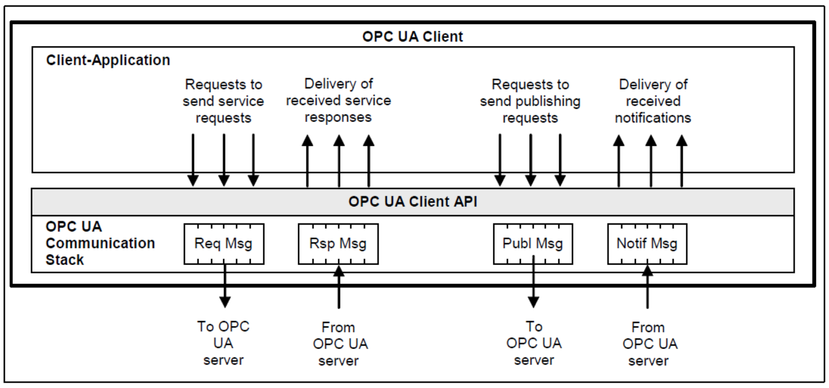
OPC UA Client
Client-Application
Requests to send service requests
Delivery of received service responses
Requests to send publishing requests
Delivery of received notifications
OPC UA Client API
OPC UA Communication Stack
Req Msg
Rsp Msg
Publ Msg
Notif Msg
To OPC UA server
From OPC UA server
To OPC UA server
From OPC UA server

Figure 2.1: General architecture of an OPC UA client illustrating all the major components [17])

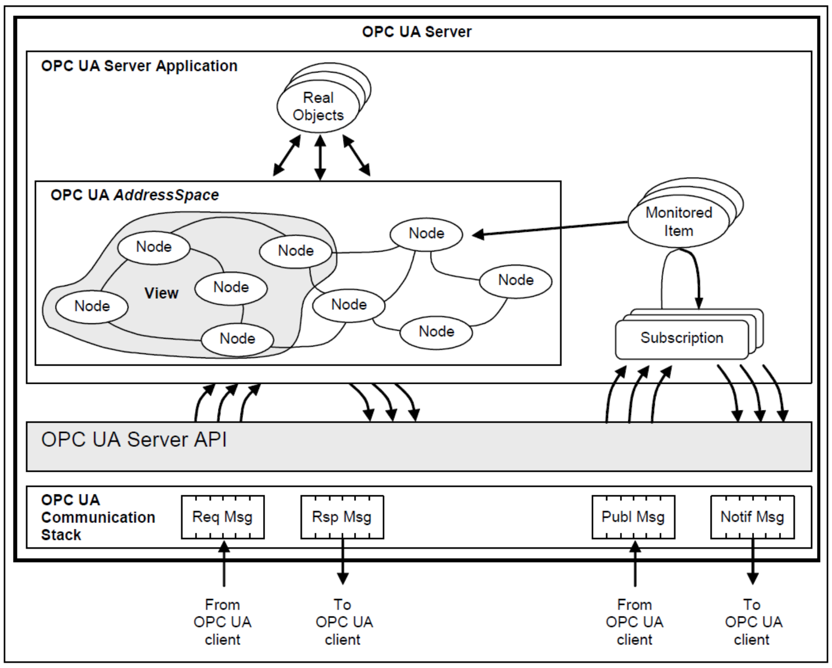

Figure 2.2: General architecture of an OPC UA server illustrating all the major components [17]

According to the specification, OPC UA defines two different types of client-server interactions. First, clients may send asynchronous service requests. These requests are used to access certain nodes or fractions of the address space based on specified criteria. For this purpose, OPC UA servers are able to provide views which define a subset of the overall address space. This allows for a more efficient service implementation as only a relevant fraction of the address space has to be analyzed. Second, clients are able to create monitored elements called MonitoredItems within the server application allowing them to monitor specific or a certain set of nodes. A client may subscribe to any type of notification like alarm notifications for selective MonitoredItems using the subscription mechanism of the server application. Thus, if an event is triggered or an attribute changes its value, the subscription mechanism propagates this change of state to all subscribed clients in form of notification messages. In addition, clients may also subscribe to periodic status updates of a node or a set of nodes. The rate these messages are sent can be configured by the subscribed client individually.

But as standard client-server communication is only based on a horizontal level, a server-to-server communication has to be established to also enable a vertical data exchange. An interlinked server structure with a bidirectional communication between two servers of different levels is required. This means that a server connecting two levels acts as a server on its own level but as a client on the other level. In order to satisfy the extensive requirements of the specification, OPC UA provides many sets of services to offer a large number of functional flexibility while also ensuring data consistency, integrity, scalability, security and redundancy.

2.3 DDS

The *Data Distribution Service* (DDS) [18] represents an industrial standard which relies on the publish/subscribe communication model and guarantees an efficient and high-performance approach to exchange information between information producers and consumers in distributed applications. In the context of the publish/ subscribe communication model, an information producer is called a publisher and an information consumer is called a subscriber. The DDS communication model operates data-centric meaning that the primary focus of the system relates to the data itself. For that purpose, any subscriber knows the type of data to be expected from any publisher as the type of data to be transmitted is already pre-defined. Furthermore, DDS guarantees high-performance and data consistency. Thus, once a publisher transmits data into the global data space of a DDS domain, all applications which are interested in that specific type of data instantaneously have access to it. A data-centric publish/subscribe communication model has the advantage that any selective data from a vast pool of information can specifically be provided for any application within the network, while publishers and subscribers are strictly decoupled from each other. Figure 2.3, adapted from [19] shows a general overview of the DDS architecture including all its major components.

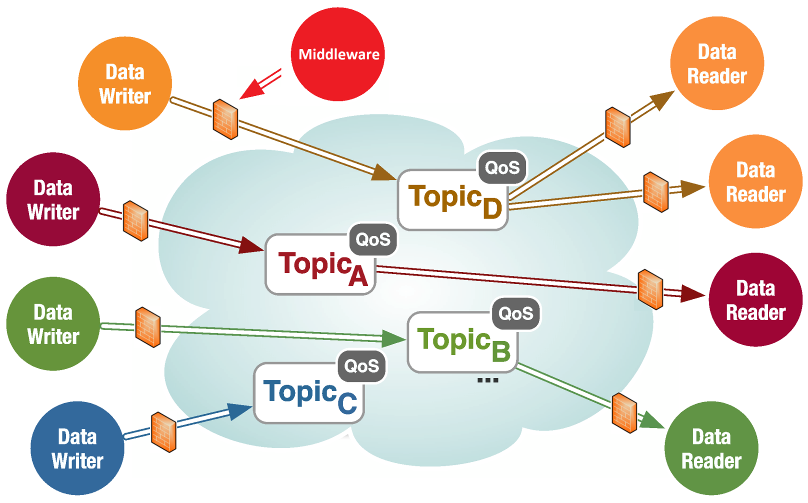

Figure 2.3: General architecture of the DDS system illustrating all the major components. The cloud represents the global data space of a DDS domain

The core of the DDS concept is the middleware which creates a virtual global data space by propagating any data published by a publisher to any subscriber which subscribed for that type of data. The middleware provides a standardized interface for application portability and ensures an efficient, robust, predictable and secure exchange of information.

Additionally the middleware precisely defines the communication routine and semantics between the publishers and subscribers as well as the integrated API. To meet these requirements, the DDS middleware uses typed interfaces and further relies on Quality of Service (QoS) measures. Typed Interfaces specify the structure and the actually used data types of the data to be transmitted. This allows to guarantee a certain level of type safety. On the other hand, QoS defines the behaviour of the service and the communication excellence from the perspective of the application. Several QoS policies describe what properties have to be implemented to enable the middleware to pre-allocate resources and assign them to the most critical sections to efficiently fulfill real-time communication requirements. Moreover, the specification requires the separability between publishers and subscribers, which means that an application only needs to implement the appropriate part of the communication participant it intends to represent. The strict decoupling of the publisher and subscriber role leads to a higher flexibility and scalability concerning the number of participants within the network.

The principle of the Data-Centric Publish/Subscribe (DCPS) model builds on the data model which specifies how distinct parts of the global data space are accessible and which data structures are used to represent the necessary information. In order to uniquely identify each portion of the global data space, an identifier in the format of a topic is introduced.

The functionality an application can implement to publish and subscribe to any data objects via their topics is specified through the Data-Centric Publish-Subscribe Platform Independent Model (DCPS PIM). The DCPS PIM implements the concept of the DDS specification using the UML modeling language where modeling is based on classes. These classes are characterized by several attributes and operations. Attributes consist of names and types while operations define an operation name, a return type and several parameters and parameter types.

The general conceptual outline specifies data objects which are identified by a topic identifier, the constructs publisher and data writer for the producer and the constructs subscriber and data reader for the consumer. The publisher represents an object which distributes information to one or many subscribers. The data writer represents the connection between the publisher and the producer application. The subscriber represents an object which receives information from one or more publishers. The data reader equals the interface between the subscriber and the consumer application. These abstractions are necessary to provide a portable, standardized and typed representation of the transmitted data independent of the application. In addition, all DCPS communication entities are part of a domain participant. The domain participant specifies to which logical domain a distributed application is assigned as DDS may be organised in more than one logical domain. Thus, publishers and subscribers are bound to a domain and may only interact if they are members of the same logical domain (even if they are members of the same physical network).

The description of the DCPS PIM is further composed of five modules. The infrastructure module defines all the abstract classes and standardized interfaces implemented by the other modules. The domain module represents all classes and interfaces needed for the organisation and the management of the domains. The topic-definition module specifies all components necessary for the definition of topics. The publication module and the subscription module define all the relevant classes and interfaces needed for the publication and subscription of any data.

2.4 OPC UA/DDS Gateway

Both industrial standards, OPC UA and DDS, which are located at the SCADA level of the automation pyramid are used to provide an efficient, robust and scalable infrastructure for the IIoT [8]. As each standard is based on another concept and communication model, both have many individual advantages but also weaknesses. Moreover, there are many differences between these approaches. OPC UA offers a common information model of the industrial automation and relies on a strong client-server architecture where clients can access resources mapped by the address space of the server using many different sets of services. The general concept is based on a Remote Procedure Call (RPC) meaning that the client invokes a service which is executed on the server manipulating or navigating through the address space and returning some result. The client-server communication model tightly couples each client with a server leading to a pure one-to-one communication scheme. On the contrary, DDS defines a data-centric publish/subscribe communication model, where the middleware distributes information provided by producers, also called publishers, to consumers, also called subscribers. This approach strictly separates both communication ends, offering a one-to-many or many-to-many communication as well as a vastly scalable network architecture.

In order to combine the benefits of both concepts, OPC UA and DDS have to be efficiently connected and integrated into a common system architecture. On large scale, DDS is used to guarantee scalability, performance and a global data space, while OPC UA is used in form of multiple subsystems to provide a standardized information model and data exchange for all low-level and field devices. But to realize this common system architecture, OPC UA and DDS have to agree on mutual communication semantics which is quite difficult between two distinct communication models. The first step towards this goal is made by the DDS family of standards when adding the RPC over DDS specification. This extends the DDS system to support RPC by using a RPC framework which enables request-reply semantics via its own native communication constructs.

With this extension of the DDS system, a standardized, interoperable, vendor-independent and configurable gateway can be specified to enable a performant and robust communication between OPC UA and DDS. The specification of the OPC UA/DDS Gateway defines two separate functional components. These functional components represent bridges which connect the OPC UA and the DDS domain but operate unidirectionally based on the intended information control flow between both domains. The OPC UA to DDS bridge ensures that DDS applications can communicate with different OPC UA servers, navigate through their address space, manipulate nodes and data within nodes and fetch information. Complementarily, the DDS to OPC UA bridge allows OPC UA client applications to participate in the global data space of the DDS system as a publisher and/or a subscriber.

The following two subsections describe the OPC UA to DDS bridge and the DDS to OPC UA bridge in more detail.

2.4.1 OPC UA to DDS Bridge

The OPC UA to DDS Bridge 2.4 establishes the communication between DomainParticipants of the DDS domain on the one side and OPC UA servers of the OPC UA system on the other side. With the integration of the OPC UA to DDS Bridge, DDS applications are able to access (read, write, modify) and receive notifications on resources provided through the address space of an OPC UA server. Hence, the OPC UA to DDS Bridge must provide one interface for each architecture to connect them. The DDS endpoints integrated in the OPC UA to DDS Bridge act as an interface for DomainParticipants of any DDS domain. The DDS endpoints participate in any DDS domain and are able to publish and subscribe to topics and receive notifications on these topics. The OPC UA clients integrated in the OPC UA to DDS Bridge act as an interface for OPC UA servers of the OPC UA system. These OPC UA clients are part of the OPC UA system and interact with OPC UA servers like any other ordinary OPC UA clients using standardized OPC UA request-reply semantics.

When a DDS application wants to interact or communicate with an OPC UA server, it sends its request to a DDS endpoint. The OPC UA to DDS Bridge then forwards this request to an OPC UA client which forwards the request to the appropriate OPC UA server. Upon completion, the OPC UA server sends a response to the OPC UA client which is forwarded to the DDS endpoint and finally back to the original DDS application.

In order to efficiently and correctly handle incoming requests and responses, the OPC UA to DDS Bridge defines three types of mappings to appropriately map all types of interactions and communications taking place. First, OPC UA type system to DDS type system mappings. These generally correspond to syntax mappings including data types, array definitions and structural mappings. Second, OPC UA Service Sets to DDS Service Sets mappings. These map the service routines of the OPC UA domain to the equivalent service routines of the DDS domain and vice versa. Third, OPC UA Subscription to DDS Subscription mappings so that any DDS application may also subscribe to data items via MonitoredItems within an address space of an OPC UA server. The OPC UA/DDS Gateway specification also defines a configuration interface for these mappings, so that any types of mappings can directly be adapted and configured to the immediate requirements of the user.

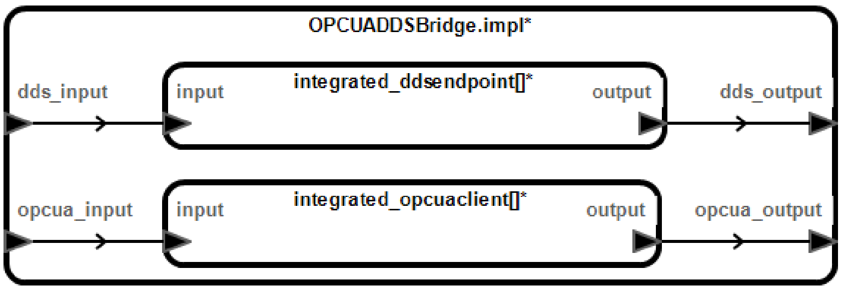

Figure 2.4: Structure diagram of the AADL system model for the OPC UA to DDS Bridge

2.4.2 DDS to OPC UA Bridge

The DDS to OPC UA Bridge 2.5 equals the complementary part of the OPC UA to DDS Bridge and establishes the communication between OPC UA clients of the OPC UA system on the one side and DomainParticipants of the DDS domain on the other side. With the integration of the DDS to OPC UA Bridge, OPC UA client applications are able to access (read, write, modify) and receive notifications on resources provided through the global data space of any DDS domain. Thus, the OPC UA/DDS Gateway must provide one additional interface for each architecture to connect them via the DDS to OPC UA Bridge. The DDS endpoints integrated in the DDS to OPC UA Bridge further act as an interface for DomainParticipants of any DDS domain. For the OPC UA system, an OPC UA server acts as an interface for the OPC UA clients of the OPC UA domain. The OPC UA server interface replicates the global data space of any DDS domain using the nodes and references of its address space. In this way, OPC UA client applications may either use the View Service Set, the Subscription and MonitoredItems Service Set or the Attribute Service Set to access the global data space of any DDS domain and publish and/or subscribe to topics like any other ordinary DDS applications.

When an OPC UA client application wants to interact or communicate with a DDS application, it sends its request to the OPC UA server of the DDS to OPC UA Bridge. The DDS to OPC UA Bridge then forwards this request to a DDS endpoint which forwards the request to the appropriate DDS application. Upon request completion, the DDS application sends a response to the DDS endpoint which is forwarded to the OPC UA server of the DDS to OPC UA Bridge and finally back to the original OPC UA client application.

In order to efficiently and correctly handle incoming requests and responses, the DDS to OPC UA bridge defines two different types of mappings to appropriately map all types of interactions and communications. First, DDS type system to OPC UA type system mappings. These represent the counterpart to the syntax mappings used for the OPC UA to DDS Bridge. Second, OPC UA specifies a precise information model which is applied to correctly represent the global data space of the DDS system within an address space of an OPC UA server. Similarly to the OPC UA to DDS Bridge, the OPC UA/DDS

Gateway specification also defines a configuration interface for the mappings of the DDS to OPC UA Bridge so that any types of mappings can directly be adapted and configured to the immediate requirements of the user.

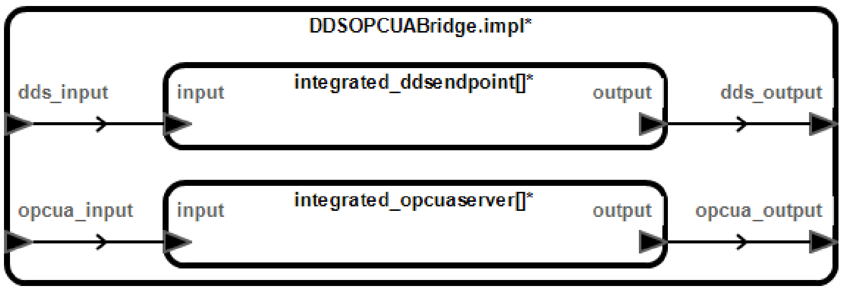

Figure 2.5: Structure diagram of the AADL system model for the DDS to OPC UA Bridge

2.5 Information Models

In order to appropriately model all the communication semantics needed by the OPC UA/DDS Gateway for the correct mapping of OPC UA syntax, semantics and services to DDS syntax, semantics and services and vice versa, precise information semantics as well as syntax mappings and interaction relations are required to be included into some standardized model. In general, information models fill this gap by specifying well-defined communication semantics, individual interaction processes and an arbitrary number of relationships between several different components and the syntax used to represent these components. One realization of such an information model for software architectures is provided by the UML which allows to model different software components and their relationships into an overall model of the common system. In this context, UML is a very powerful modeling language as several aspects of the software engineering process (specification, documentation, analysis, etc.) are considered. But in the presence of safety-critical real-time embedded systems that mostly rely on tasks to complete within hard deadlines, where additional hardware modeling support is required to guarantee the specification of the system (avionics systems in aircrafts for instance), the boundaries of UML are exceeded. For this purpose, a modeling language is needed which allows the modeling of software components and their relationships on the one hand, but also the respective computer architectures, their individual hardware components and all their properties that apply when the system is deployed on the other hand. For achieving hardware modeling and real-time analysis support, an extended information modeling language called AADL is introduced.

2.6 AADL Modeling

The *Architecture Analysis & Design Language* (AADL) allows to define a system model which considers the entire software architecture with all its components and their relationships while also providing hardware modeling, safety and real-time analysis support [10]. This enables AADL to additionally model physical components of a system and further satisfy the requirements of a real-time embedded systems model. Moreover, AADL supports availability & reliability analysis, security analysis, data quality and real-time performance analysis as well as resource consumption analysis. In synergy with the information model of the software architecture, each aspect of a system may be analyzed to find any possible exploits or flaws within the overall system that may cause severe problems later in the development cycle of the system or once the system is deployed. The necessity of this comprehensive analysis is best shown as the later case could not only result in costly counter-measures but also catastrophic outcomes leading to the physical destruction of system components, ecological or collateral damage or even human deaths depending on the type of application the system is responsible for. In order to guarantee all of these requirements and still enable an efficient and agile modeling process, AADL builds on a hierarchical system model and splits all of its components into three categories: software, hardware and hybrid components (also called composite components). This allows an individual, clear and performant modeling of both the software and the hardware domain while the analysis can either be performed for each model individually or for the overall system as a whole. The modeling process in AADL may be performed textually and/or graphically. Thus, AADL can be used like any ordinary high-level programming language while the system model can also be graphically illustrated and modified to maintain a better overview of the system model. In order to get familiarized with the AADL syntax though, the AADL modeling basics explained in this section are described by textually represented system models.

The basic concept of AADL relies on a hierarchical system model with one top-level system component where each component may have several subcomponents and a system model may be a subsystem of another hierarchically higher system model. Analog to the object-oriented paradigm, components are able to inherit features and properties from their parent. AADL also specifies a certain level of abstraction as each component defines a type and an implementation separately. The type of a component identifies the representation of that component and how it may interact with other components (interfaces, etc.). The implementation of a component defines its functionality and internal structure. Therefore, the implementation of a component represents the realization of the component's specific type. Though components are restricted to only one type, a component may have several implementations related to that type. Besides having an arbitrary number of subcomponents, components are also able to communicate with other components using features and connections to define different types of component communication. Furthermore, AADL allows to specify properties for each type of software, hardware or hybrid component. These properties enable the additional specification and application of assumptions related to real components to guarantee a more realistic analysis of the system once it is deployed. For software components, such properties may be the dispatch protocol used, the information control flow of a connection between components or the period of some thread that is executed for instance. Related to hardware components, timing requirements, scheduling algorithms, power consumption of specific hardware units or bus bandwidth and processor frequency are certain points of interest for example. However, the precise specification of these properties for any type of software or hardware component is explained in Section 2.6.2 in more detail.

2.6.1 Modeling Basics

Regarding the modeling process, all software, hardware and composite components which intend to be part of the system model are modeled as subcomponents of the system implementation. The generic syntax to model some component as a subcomponent of a system model is shown by code Listing 2.1 below.

```
system <System name> // type definition of the system model
end <System name>;

system implementation <System name>.<System implementation name> //
    implementation of the system model
        subcomponents
                <internal component name>: <software/hardware type
                    identifier> <external component name>;
end <System name>.<System implementation name>;
```

Listing 2.1: Generic definition of a system model integrating a subcomponent

In order to provide a concrete example on how to define a component and specify its type and implementation, Listing 2.2 shows the component type and implementation of some exemplary system model which has a subcomponent called Controller. Assume that the Controller device component is already defined (the modeling of hardware components is explained in *Section 2.6.3*).

```
system ExampleSystem // type definition of the system model
end ExampleSystem;

system implementation ExampleSystem.including_controller //
    implementation of the system model integrating the Controller
    device
        subcomponents
                sys_controller: device Controller; // device refers
                    to some hardware device named Controller
end ExampleSystem.including_controller;
```

Listing 2.2: Definition of a device named Controller as subcomponent of the system model

AADL is case-insensitive meaning that each name declaration and definition is uniquely identified by its writing. Additionally, AADL models may be structured into one or many packages (libraries, source files, etc.) where component types and implementations can be outsourced and later reintegrated into other components, subsystems or the system model by importing the corresponding packages and referring to the components within those packages. The name space of a package is divided into public and private segments though. A component in the public segment of the package may be accessed from another package using that package as a reference to that component. A component in the private segment of the package must not be accessed from another package as it is only visible within the package its type and implementation are defined. Although software, hardware and composite components are basically modeled within the same top-level system model, the following subsections individually refer to the modeling basics of software components 2.6.2, hardware components 2.6.3 and composite components 2.6.5 in AADL to provide a better overview.

2.6.2 Modeling Software Components

AADL defines four different types of components that may be used to identify and represent entities which occur in the software architecture of a system: processes, threads, data and subprograms.

Equally to processes within any operating system, processes in AADL represent executable application instances assumed to run on a processor. Therefore, components of the process type model processes of a real system within the system model. Similarly, AADL thread type or thread group type components model threads or groups of threads of a real system within the system model.

The following Listing 2.3 shows how a process component and a thread component can be modeled within the system model.

```
	system ExampleSSystem // type definition of the system model
	end ExampleSSystem;

	system implementation ExampleSSystem.processes_threads // implementation of the system model integrating a process unit and a thread unit
		subcomponents
			sys_process: process P1.impl; // process refers to some process component named P1
	end ExampleSSystem.processes_threads;

	process P1 // type definition of P1
	end P1;

	process implementation P1.impl // implementation of P1
		subcomponents
			t1: thread T1; // thread T1 as subcomponent of process P1
	end P1.impl;

	thread T1 // type definition of T1
```

```
        end T1;
```

Listing 2.3: Definition of a process unit named P1 and a thread unit called T1 as subcomponents of the system model

Compared to threads and processes in an operating system, thread type components are supposed to be part of process type components and represent individual executables for specific application purposes. For example, thread T1 in Listing 2.3 may refer to reading and monitoring a sensor node and communicate its value. For this purpose, different processes and threads have to share and communicate information which yields the need for a common interface. Thus, AADL specifies two communication constructs to communicate between components in form of features and connections.

Features provide a common communication link where one feature uses a port as interface for example that triggers either on an event and/or consistently communicates data depending on the type of communication which is intended. Supplementary, it must be specified, if the communication is incoming or outgoing. A feature is identified by a feature name and has the following definition syntax in a component:

<feature name>: <in/out> <data/event/eventdata> port;

The syntax shown above is used to define a feature by a port. Furthermore, features may also use component access, subprogram calls and parameter interfaces (connectors) instead of ports to enable communication between components.

Connections provide a common communication construct which defines the mapping of one communication interface to another communication interface (one feature to another feature for example). Thereby, each connection is defined by a connection name and has the following syntax when used in combination with feature ports:

<connection name>: port <component name>.<outgoing port name> -> <component name>.<incoming port name>

Connections can also be specified for buses where the syntax is similar to the syntax of mapping feature ports (described in *Section 2.6.3* in more detail). Analog to features, connections define many interfaces used to establish links between components. These are identified by the data, event, eventdata, dataaccess, busaccess and port classifier.

For illustrating how the communication between two processes (or threads) works, Listing 2.4 shows the process, thread, feature and connection definitions required for process P1 to communicate with thread T1 and for thread T1 to communicate with thread T2.

```
        system ExampleSSystem // type definition of the system model
        end ExampleSSystem;

        system implementation ExampleSSystem.features_connections //
            implementation of the system model integrating a process unit and
             two thread units
                subcomponents
            sys_process: process P1.impl; // process refers to some process component named P1
    end ExampleSSystem.features_connections;

    process P1 // type definition of P1
        features
            T1_event: in event port; // wait for event
    end P1;

    process implementation P1.impl // implementation of P1
        subcomponents
            P1_thread1: thread T1; // name of thread T1 in P1.impl
            P1_thread2: thread T2; // name of thread T2 in P1.impl
        connections
            connect_P1T1: port T1_event -> P1_thread1.T1_event; // forward input from process P1 to thread T1
            connect_T1T2: port P1_thread1.send_value -> P1_thread2.read_value; // forward output from thread T1 to thread T2
    end P1.impl;

    thread T1 // type definition of T1
        features
            T1_event: in event port; // wait for event
            send_value: out data port; // propagate data
    end T1;

    thread T2 // type definition of T2
        features
            read_value: in data port; // consume data
    end T2;
```

Listing 2.4: Established communication between two thread units named T1 and T2 via features and connections

In the example system model shown by Listing 2.4, the event triggered input provided by feature T1_event of process P1 is forwarded to the event triggered input provided by feature T1_event of thread T1 using the connection connect_P1T1. In general, this means that any input received by P1 on feature T1_event is directly forwarded to T1's corresponding input feature. Equally, any data output produced by feature send_value of T1 is directly forwarded to be consumed by feature read_value of T2 using the connection connect_T1T2. This demonstrates that by extending the system model with features and connections, AADL allows to model interthread and interprocess communication for the software model of the system regarding any certain type of application context.

However, besides features and connections, AADL further allows the definition of an information flow for any link between one or more components using the flows operator. In this way, an information producer called a source and an information consumer called a sink can be specified for the corresponding interfaces. This provides enhanced analysis capabilities as the particular flow of data is well defined which is especially important for analyzing the communication and interaction of different components.

AADL also allows to model data type components which act as data placeholders and represent static data records in a system model. Like any other component type, data type components may be composed of several data type subcomponents resulting in the mapping of hierarchical data models that allow to distinct between generic data type components and more specific data type components. Exemplary, AADL data models can be compared to structs of the C/C++ programming language. Analog to structs, data type components can specify one or more internal data fields (primitive or nested data type components) of which the data record is composed of using the properties or the subcomponent operator. For this purpose, AADL provides a set of supported built-in data types that can be used to represent scalars, arrays or sequences of primitive data types. Listing 2.5 shows the data model of a structure in AADL.

```
        data NodeId // generic data type component
                properties
                        Data_Model::Data_Representation => Struct; //
                            represents structure
        end NodeId;

        data implementation NodeId.impl
                subcomponents
                        namespace_index: data Base_Types::Unsigned_16;
                        identifier_type: data NodeIdentifierType; // some
                            data model of an enumeration type
        end NodeId.impl;

        data ExpandedNodeId extends NodeId // specific data type component
            which extends the generic data type component
                properties
                        Data_Model::Data_Representation => Struct; // also
                            represents a structure
        end ExpandedNodeId;

        data implementation ExpandedNodeId.impl
                subcomponents
                        namespace_uri: data Base_Types::String;
                        server_index: data Base_Types::Unsigned_32;
        end ExpandedNodeId.impl;
```

Listing 2.5: Definition of a hierarchical data model. The generic data type component is called NodeId while the specific data type component is called ExpandedNodeId

Besides the common structure data model, data type components alternatively allow to represent other complex types like enumerations, arrays and unions. Listing 2.6 illustrates the data type component NodeClass which represents an enumeration data type.

```
        data NodeClass // type definition of the data type component
        properties
                Data_Model::Data_Representation => Enum; //
                    represents enumeration
                Data_Model::Enumerators => ("OBJECT_NODE_CLASS", "
                    VARIABLE_NODE_CLASS", "METHOD_NODE_CLASS",
                "OBJECT_TYPE_NODE_CLASS", "VARIABLE_TYPE_NODE_CLASS",
                     "REFERENCE_TYPE_NODE_CLASS", "
                    DATA_TYPE_NODE_CLASS",
                "VIEW_NODE_CLASS"); // specify enumerators
    end NodeClass;
```

Listing 2.6: Definition of an enumeration data type component called NodeClass

Moreover, subprograms or subprogram groups (later indicate libraries of subprograms) may also be specified by components in AADL. Although the actual behaviour of a subprogram is represented by the semantics of the source code of the program or the model, AADL allows the integration of subprograms into the system model and enables the analyses of these subprograms in the system model to define which impact they enforce on the software model and the overall requirements of the system. In order to model a subprogram in AADL, certain properties like the implementation language, the function or method name and the source file have to be stated.

Similar to the modeling of connections and features, properties may also be integrated into components to approximate their (physical) constraints and boundaries, define additional parameters and to make the overall system model more realistic and better analyzable compared to the real system. Although the properties operator is explained within this section in more detail, properties may also be specified for hardware components, composite components or even the system model as already stated earlier.

Applied to a general component type definition, any property specified is defined by the following syntax shown below.

<property name> => <property value>;

Listing 2.7 shows an example usage of the properties operator where a C subprogram function CheckSensor with the source file name CheckSensor.c is called by a thread named T1 on each periodic dispatch.

```
    subprogram CheckSensor // subprogram component representing a C
        function
        features
                value: in parameter sensor_reading; // function
                    CheckSensor has one input parameter
        properties
                Source_Language => (C); // source language is C
                Source_Name => "CheckSensor"; // name of the function
                Source_Text => ("CheckSensor.c"); // name of the
                    source file
    end CheckSensor;

```

```
thread T1 // type definition of thread T1
        features
                T1_value: in data port sensor_reading;
        properties
                Dispatch_Protocol => Periodic; // periodic dispatch
                Period => 1000ms; // period of the thread
                Deadline => Period; // relevant for the scheduling of the processor
                Priority => 1; // relevant for the scheduling of the processor
                Compute_Execution_Time => 1ms .. 2ms; // execution time of T1 is estimated between 1ms and 2ms
end T1;

thread implementation T1.impl // implementation of Thread T1
        calls C : { // calls are executed at each dispatch
                call_sensor: subprogram CheckSensor; // specify that subprogram CheckSensor is executed at each dispatch
        };
        connections
                sensor_par1: parameter T1_value -> CheckSensor.value; // map thread input parameter to subprogram input parameter
end T1.impl;
```

Listing 2.7: Definition of a C subprogram function CheckSensor with the source file name CheckSensor.c and a thread unit called T1

Last but not least, AADL defines modes which correspond to the state of a component, a subsystem or the system model. The mode of a component may affect the integration of subcomponents, the existence of connections or specific property values. The transition of a mode to another state may be indicated by the occurrence of a received event followed by a reconfiguration based on that event.

2.6.3 Modeling Hardware Components

In order to model the physical components of a real system, AADL specifies four distinct types of components that may be used to identify and represent entities of the hardware architecture of a system: processors, buses, memories and devices.

Based on the common understanding of processors, processor components modeled in AADL are hardware components which represent processor units of a real system within the system model. Analog to the processor component, memory components equally represent memory units of the real system within the system model. Listing 2.8 shows how a processor unit and a memory unit of a real system can be integrated into the system model.

```
system ExampleHSystem // type definition of the system model
end ExampleHSystem;
```

```

        system implementation ExampleHSystem.processor_memory //
           implementation of the system model integrating a processor unit
           and a memory unit
                subcomponents
                        sys_processor: processor P1.impl; // processor refers
                             to some processor component named P1
        end ExampleHSystem.processor_memory;

        processor P1 // type definition of P1
        end P1;

        processor implementation P1.impl // implementation of P1
                subcomponents
                        P1_memory: memory M1;
        end P1.impl;

        memory M1 // type definition of M1
        end M1;
```

Listing 2.8: Definition of a processor unit named P1 and a memory unit called M1 as subcomponents of the system model

The memory unit M1 can be modeled as part of P1's processor implementation as the memory unit may be assumed to be a subcomponent of the processor while P1 represents a subcomponent of the system model. As illustrated in Listing 2.8, M1 is a subcomponent of P1 which is a subcomponent of the system model, thus M1 is transitively also a subcomponent of the system model.

But according to the hardware architecture of a real system, more hardware components are present than processor cores and memory units. In order to identify and represent these components in the system model as well, the device type identifier and the bus type identifier are introduced. Compared to any classical hardware architecture, the device type identifier generally represents any hardware component which corresponds to an Input/Output (IO) device. The bus type identifier enables a component to act as hardware bus and for establishing the communication between all hardware components which are connected to the bus. For instance, some processor component may communicate with one or many device components.

In order to enable the bus access for any hardware component, four configuration steps are required. First, the bus type component has to be integrated as a subcomponent of the system model. Second, the hardware component that wants to access the bus has to be integrated as a subcomponent of the system model. Third, any hardware component that wants to access a bus has to define which bus it wants to access. This can be specified within the features interface by the following syntax:

<feature name>: required bus access <bus name>;

Fourth, the connection between the bus component and the corresponding hardware component has to be mapped by a connection name within the system model. In order to grant bus access to any hardware component identified by the system model, the following syntax is provided:

<connection name>: bus access <internal bus name> -> <internal component name>.<feature name>;

In order to get a better understanding for the modeling of hardware buses in AADL, Listing 2.9 shows how to integrate a bus type component and a device type component into the existing system model of Listing 2.8. In this case, processor P1 and device Dev1 are both granted access to bus HWBus.

```
system ExampleHSystem // type definition of the system model
end ExampleHSystem;

system implementation ExampleHSystem.processor_device_bus // implementation of the system model integrating a processor unit, a memory unit, a device unit and a bus unit
        subcomponents
                sys_processor: processor P1.impl; // processor refers to some processor component named P1
                sys_device: device Dev1; // device refers to some device component named Dev1
                sys_bus: bus HWBus.impl; // bus refers to some bus component named HWBus
        connections
                bus_processor: bus access sys_bus -> sys_processor.bus_access; // grant bus access to processor P1
                bus_device: bus access sys_bus -> sys_device.bus_access; // grant bus access to device Dev1
end ExampleHSystem.processor_device_bus;

processor P1 // type definition of P1
        features
                bus_access: requires bus access HWBus; // P1 wants to access bus HWBus
end P1;

processor implementation P1.impl // implementation of P1
        subcomponents
                P1_memory: memory M1;
end P1.impl;

memory M1 // type definition of M1
end M1;

device Dev1 // type definition of Dev1
        features
                bus_access: requires bus access HWBus; // Dev1 wants to access bus HWBus
end Dev1;

device implementation Dev1.impl // implementation of Dev1
```

```
        end Dev1.impl;
```

Listing 2.9: Established communication between device Dev1 and processor P1 via bus HWBus

2.6.4 Binding Software and Hardware

A primary benefit of AADL is the capability to model and analyze the interaction of the software architecture and the hardware architecture within one common system model. With the introduction of binding properties, components of the software architecture can be mapped to components of the hardware architecture and specifically assigned usable resources. For this purpose, three types of binding properties exist: processor binding, memory binding and connection binding.

The Processor Binding property defines the precise scheduling and execution of selected processes and threads on a processor.

The Memory Binding property defines which memory components are particularly responsible for storing selective processes, threads and their data.

The Connection Binding property precisely defines the physical communication channels used for the existing logical connections.

2.6.5 Modeling Composite Components

In extension to the classical models of software and hardware components of a system model, AADL further allows to describe components which are neither identified as pure software components or pure hardware components at the time of definition and may even be composed of several subsystem models. These components are called hybrid or composite components and are introduced by AADL to make the system model even more realistic compared to the real system. The benefit of these components is that they allow to combine software and hardware components to guarantee a more flexible modeling process in terms of software/hardware architecture interaction as real system subcomponents may also consist of software and hardware parts. The term composite, hybrid or also called abstract or generic is not a type identifier of a component definition or implementation though, rather than a concept to improve the modeling capabilities of the AADL.

CHAPTER 3

Scientific Methodology

The research method applied in this thesis follows a design and creation research strategy and builds upon academic literature as well as relevant specifications [20]. In computing research, the design and creation strategy focuses on developing new IT products called artefacts [21]. Such artefacts include constructs, models, methods and instantiations [22], whereby in most research, a combination of several artefacts contribute in creating new knowledge. As artefacts often represent computer-based products, the design and creation research strategy emphasises on analysis, explanation, argument, justification, and critical evaluation of the results to distinguish itself from product development.

In the context of design and creation research, the focus lies either on the artefact itself, (e.g. the IT application incorporates a new theory), the artefact as a vehicle to create new knowledge (e.g. the IT application in use) or on the process to create an artefact to create knowledge [23]. Within this thesis, the focus is two-fold and lies in the creation process as well as on how the artefact performs in its application domain. Specifically, the project creates knowledge in applying the AADL modeling language and insights on configuration challenges related to the OPC UA/DDS Gateway.

Design and creation research is a problem-solving approach and builds upon the principles of system development [24]. The process involves typically five steps—awareness, suggestion, development, evaluation and conclusion. Whereby the steps are not rigid in order instead form an interative cycle. As a significant effort in computer-based research often concentrates in the development step, it is necessary to apply a specific system development method (analysis, design, implementation and testing). This research project uses the UML systems development methodology [25], as it covers modeling techniques.

Figure 3.1 visualises the research method and the underlying applied strategies and data sources applied in this research project.

The following paragraphs give a short introduction about the used tools respectively applications. The first described tool is the open-source toolchain OSATE which is the

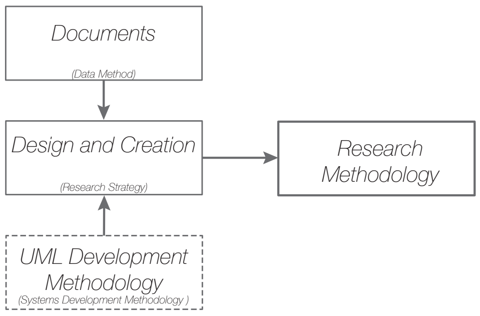

Figure 3.1: Research method overview and elements based on [20]

Integrated Development Environment (IDE) used for the creation of the AADL system model for the OPC UA/DDS Gateway. The Ocarina toolchain allows generating C or Ada code from the specification of an AADL system model.

Chapter 4 presents further details about the modeling process of the OPC UA/DDS gateway and *Chapter 5* the results related to the gateway configuration discussion.

3.1 OSATE

OSATE is an open-source tool platform [14] supporting the modeling and analysis process of real-time system models using AADL. It allows to view and edit the AADL system model textually as well as graphically. While the graphical editor provides a very high-level abstraction of the system model by illustrating all its major components, their features and the connection between those components in form of a variety of diagrams, the textual editor enables a more detailed in-depth representation and configuration of the AADL system model e.g. individual specification of certain properties. Furthermore, OSATE is highly extensible meaning that it allows the integration of different real-time analysis tools e.g. Cheddar which is a real-time scheduling simulator. Moreover, OSATE also provides a plugin for integrating Ocarina 3.2 and generating target specific C or Ada code based on the AADL system model specification. The OSATE version that was used for modeling the OPC UA/DDS Gateway AADL system model is OSATE2-2.6.1.

3.2 Ocarina

Ocarina is an open-source toolchain [15] used for building and analyzing applications based on AADL specifications. For generating code from an AADL system model, Ocarina uses the PolyOrb-Hi-C or PolyOrb-Hi-Ada high-integrity middleware to generate POSIX conform C or Ada code. For this purpose, Ocarina provides all the functionalities necessary to parse and semantically check an AADL system model before the corresponding middleware uses its built-in constructs to translate the AADL system model description into PolyOrb-Hi-C C or PolyOrb-Hi-Ada Ada conform code. Supplementary, Ocarina also supports the feature to build applications for specific target platforms. The Ocarina version that was used to generate the C source code from the OPC UA/DDS Gateway AADL system model is Ocarina 2017.x.

3.3 Chronological Development Process

In the following, Figure 3.2 illustrates the chronological development process in form of a timeline which describes when each of the stated toolchains was primarily used.

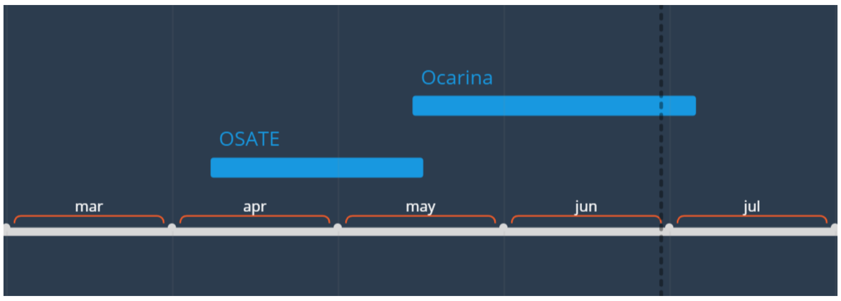

Figure 3.2: Timeline of the main toolchain usage used for development

CHAPTER 4

AADL Model for the OPC UA/DDS Gateway

The primary focus of this thesis relies on the formal AADL system model for the OPC UA/DDS Gateway specification and the dynamic configurability of the OPC UA/DDS Gateway. As the formal AADL system model lays the foundation for the dynamic configurability of the OPC UA/DDS Gateway, the latter is discussed in *Chapter 5* to maintain a better structure of the results. In comparison with the results of the scientific work stated in *Section 2.1*, this chapter of the thesis aims at providing a formal AADL system model for the OPC UA/DDS Gateway specification [8]. Furthermore, this chapter visualizes the overall modeling process, the structuring of the AADL models, the different approaches that were used in order to model certain aspects of the OPC UA/DDS Gateway specification in the AADL system model and the variety of problems that were found and solved during the modeling process. This chapter does not contain the entire code of the AADL system model for the OPC UA/DDS Gateway though, as the full code can be accessed via the GitLab project link provided in *Section 8.1*.

4.1 General Structure of the AADL system model

The OPC UA/DDS Gateway is modeled in OSATE2-2.6.1 using the AADL 2 standard. According to the description of *Section 2.4*, the general structure of the OPC UA/DDS Gateway AADL system model is shown in Fig. 4.1.

The AADL system model is composed of an array of multiple OPC UA to DDS bridges and an array of multiple DDS to OPC UA bridges which each connect the OPC UA domain with the DDS domain and vice versa, allowing requests from OPC UA clients to the DDS domain and requests (publish/subscribe) from DDS applications to the OPC UA domain to be processed and responses to be received. The advantage of having

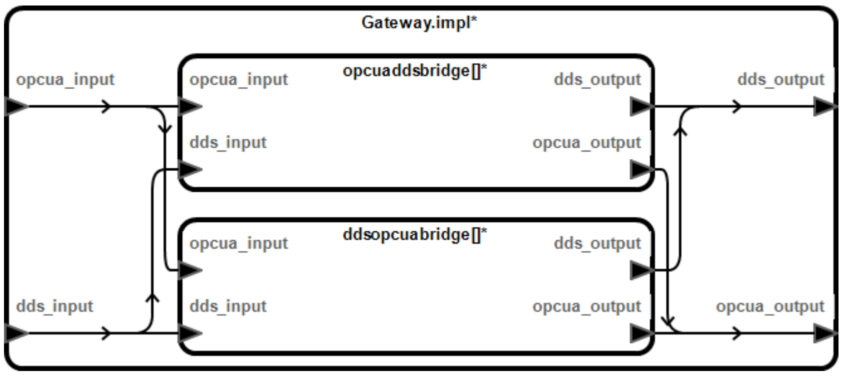

Figure 4.1: Structure diagram of the general architecture of the OPC UA/DDS Gateway AADL system model

many OPC UA to DDS Bridges and many DDS to OPC UA Bridges instead of only one Bridge each is that each Bridge may be configured individually and serve some specific application context. The following section illustrates the AADL model of the OPC UA to DDS Bridge. The AADL model of the DDS to OPC UA Bridge is demonstrated in *Section 4.3*.

Apart from the general structure of the OPC UA/DDS Gateway AADL system model, the overall AADL system model also includes several AADL models representing relevant entities of the OPC UA domain and the DDS domain. Concerning the AADL models of the OPC UA domain, the system model implements the AADL models for the OPC UA client and the OPC UA server and all the OPC UA datatype specifications. The OPC UA datatypes are distributed on multiple AADL files and categorized by OPC UA Service Set assignment if they are related to specific OPC UA Service Sets. The AADL models of the DDS domain include the general DDS datatypes specifications and the AADL model for the DDS application.

4.2 AADL Model for the OPC UA to DDS Bridge

The OPC UA to DDS Bridge 2.4.1 is modeled in AADL as shown by Fig. 4.2.

The OPC UA to DDS Bridge consists of an array of integrated DDS domain participants which each specify some generic DDS datatype as input/output and an array of integrated OPC UA clients which each specify some generic OPC UA datatype as input/output. The benefit of integrating multiple DDS domain participants and multiple OPC UA clients results in an improvement of overall performance and latency as many simultaneous requests and responses may be processed.

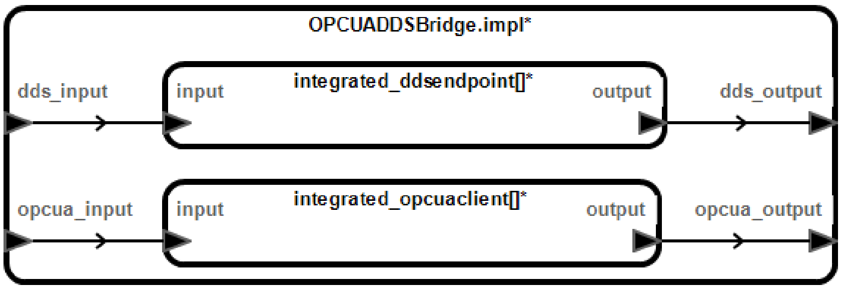

Figure 4.2: Structure diagram of the AADL system model for the OPC UA to DDS Bridge

4.3 AADL Model for the DDS to OPC UA Bridge

The AADL model for the DDS to OPC UA Bridge 2.4.2 is illustrated by Fig. 4.3.

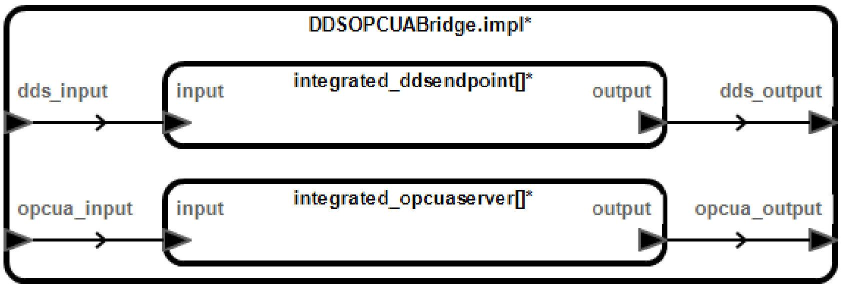

Figure 4.3: Structure diagram of the AADL system model for the DDS to OPC UA Bridge

The DDS to OPC UA Bridge consists of an array of integrated DDS domain participants which each specify some generic DDS datatype as input/output and an array of integrated OPC UA servers which each specify some generic OPC UA datatype as input/output. Analog to the OPC UA to DDS Bridge, the benefit of integrating multiple DDS domain participants and multiple OPC UA servers results in an improvement of overall performance and latency as many simultaneous requests and responses may be processed.

4.4 Modeling of OPC UA Components

The overall AADL system model of the OPC UA architecture is basically split into three AADL models: the AADL model for the OPC UA domain, the AADL model for the OPC UA client system and the AADL model for the OPC UA server system. In extension to the AADL model for the OPC UA domain, several additional AADL models for the OPC UA Service Set specific data model definitions exist.

The AADL model for the OPC UA specifies the general architecture of the OPC UA including the subcomponent specification of OPC UA clients, OPC UA servers as well as the data model definitions of all generic OPC UA datatypes and the OPC UA Service Set representations in AADL. The latter is discussed in *Section 4.4.2* in more detail. Moreover, the AADL model for the OPC UA clients is also used for the integrated OPC UA clients in the OPCUA to DDS Bridge. The AADL model for the OPC UA is shown in Fig. 4.4.

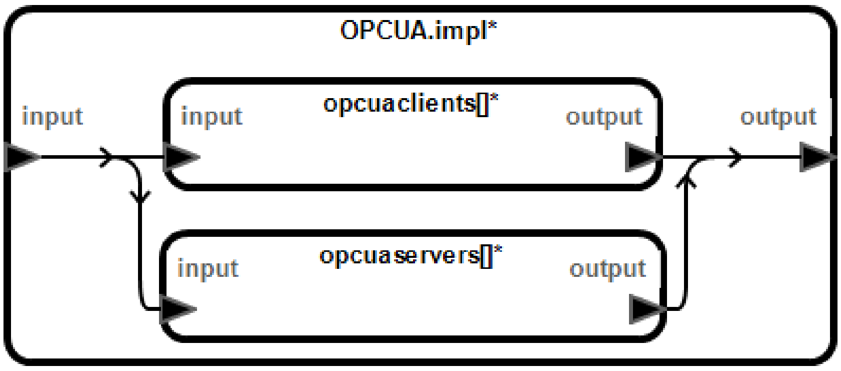

Figure 4.4: Structure diagram of the AADL system model for the OPC UA

4.4.1 Modeling of OPC UA Types in DDS

This section provides an overview on how certain component models and data models have been realized in compliance with the OPC UA/DDS Gateway specification [8] and describes the modeling approaches of OPC UA Types in DDS which are used by the OPC UA to DDS Bridges in order to establish communication and appropriately map the corresponding datatypes. Exemplary, three simple data models on how to represent structures, unions and enumerations are illustrated below. Afterwards a more complex data model is discussed.

According to the OPC UA/DDS Gateway specification, several OPC UA Types are specified as non-primitive DDS Types meaning that they cannot just be replaced with any native DDS Type. For this purpose, these OPC UA Types are represented by enumerations, unions, structures or more complex DDS Types (combination of enumerations, structures and/or unions). The following illustration of an OPC UA type to DDS type mapping refers to the FieldAssignment Type.

```
        OPC UA type                                 DDS type
----------------------------------------------------------------
        FieldAssignment         enum AssignmentKind {
                                        DATA_ITEM_ASSIGNMENT,
                                        EVENT_FIELD_ASSIGNMENT,
                                        CONSTANT_VALUE_ASSIGNMENT
                                };
                                struct DataItemRef {
                                        string data_item_name;
                                };
                                struct EventFieldRef {
                                        string event_name;
                                        uint32 event_field_index;
                                };

                                union AssignmentInput switch (AssignmentKind)
                                     {
                                        case DATA_ITEM_ASSIGNMENT:
                                                DataItemRef data_item;
                                        case EVENT_FIELD_ASSIGNMENT:
                                                EventFieldRef event_field;
                                        case CONSTANT_VALUE_ASSIGNMENT:
                                                Variant constant_value;
                                };

                                struct FieldAssignment {
                                        string dds_output_field_ref; // name
                                            of output field
                                        OpcUaInput opcua_input_ref;
                                        AssignmentInput assignment_input;
                                };
```

Listing 4.1: OPC UA type to DDS type mapping for the FieldAssignment OPC UA type

For modeling the FieldAssignment OPC UA Type in AADL, the following enumeration, union and structures have to be modeled first. The data model of the AssignmentKind enumeration type is shown below.

```
        data AssignmentKind
                properties
                        Data_Model::Data_Representation => Enum;
                        Data_Model::Enumerators => ("DATA_ITEM_ASSIGNMENT", "
                            EVENT_FIELD_ASSIGNMENT", "
                            CONSTANT_VALUE_ASSIGNMENT");
        end AssignmentKind;
```

Listing 4.2: Data model of the AssignmentKind enumeration type

The data models of the DataItemRef and the EventItemRef structure types are also shown below.

```
        data DataItemRef
                properties
                        Data_Model::Data_Representation => Struct;
                        Data_Model::Base_Type => (classifier (Base_Types::
                            String));
                        Data_Model::Element_Names => ("data_item_name");
        end DataItemRef;

        data EventFieldRef
                properties
```

```
                        Data_Model::Data_Representation => Struct;
                        Data_Model::Base_Type => (classifier (Base_Types::
                            String), classifier (Base_Types::Unsigned_32));
                        Data_Model::Element_Names => ("event_name", "
                            event_field_index");
        end EventFieldRef;
```

Listing 4.3: Data models of the DataItemRef and EventItemRef structure types

The data model of the AssignmentInput union type is already more complex, as the switch construct which is shown in the specification also needs to be integrated into the union type data model. Although there is no way of statically modeling the dynamic control flow of the switch construct in AADL, one way to still model this functionality in AADL is to integrate the AssignmentKind enumeration type as subcomponent of the union type data model and later define possible analysis constraints to approximate the dynamic behavior.

```
        data AssignmentInput
                properties
                        Data_Model::Data_Representation => Union;
        end AssignmentInput;

        data implementation AssignmentInput.impl
                subcomponents
                        assignmentkind: data AssignmentKind;
                        data_item: data DataItemRef;
                        event_field: data EventFieldRef;
                        constant_value: data opcua::Variant.impl;
        end AssignmentInput.impl;
```

Listing 4.4: Data model of the AssignmentInput union type

After the successful modeling of the previous data models, the FieldAssignment structure type can be modeled as follows.

```
        data FieldAssignment
                properties
                        Data_Model::Data_Representation => Struct;
        end FieldAssignment;

        data implementation FieldAssignment.impl
                subcomponents
                        dds_output_field_ref: data Base_Types::String; //
                            name of output field
                        opcua_input_ref: data opcua::OPCUAInput;
                        assignment_input: data AssignmentInput.impl;
        end FieldAssignment.impl;
```

Listing 4.5: Data model of the FieldAssignment structure type

4.4.2 Modeling of OPC UA Service Sets

In extension to the modeling of the appropriate OPC UA type to DDS type mapping and the basic OPC UA client and OPC UA server models, the OPC UA Service Sets that are specified by the OPC UA have to be integrated in the OPC UA AADL system model. In order to explain and state the implementation of an OPC UA Service Set in AADL, the AADL model of an OPC UA Service Set is shown by the OPC UA Subscription Service Set. The model of an OPC UA Service Set in AADL is implemented using the following approach: The general execution model of each Service Set is represented in form of a process type component which is a subcomponent of the OPC UA client system model. As all services within a service set shall be non-blocking meaning that if two clients simultaneously try to request a specific service, both requests shall be executed concurrently, thus the execution of such services or methods is modeled in form of thread type subcomponent arrays, respectively. In this way, multiple identical requests may be executed at once. The specific method realization is provided in form of subprograms which specify the parameters according to the OPC UA/DDS Gateway specification. The following Listing exemplary illustrates the abstract hierarchy of the OPC UA Subscription Service Set implementation as it is modeled in AADL.

```
                OPC UA SubscriptionServiceSetProcess // indicates the Service Set interface (process type component)
                        /                       \
        CreateSubscriptionThread[]      CreateMonitoredItemThread[] // thread type components
                /                                       \
        CreateSubscription                      CreateMonitoredItem
        // subprogram type components
```

Listing 4.6: Abstract AADL system model for the OPC UA Subscription Service Set

The concrete implementation of the OPC UA Subscription Service Set AADL model is shown below.

```
process OPCUASubscriptionServiceSetProcess // Subscription Service Set interface
        features
                output: out data port opcua::OPCUAData.impl;
                input: in data port opcua::OPCUAData.impl;
end OPCUASubscriptionServiceSetProcess;

process implementation OPCUASubscriptionServiceSetProcess.impl
        subcomponents
                create_subscription_thread: thread CreateSubscriptionThread.impl[]; // assign thread as subcomponent of client process
                create_monitored_item_thread: thread CreateMonitoredItemThread.impl[];
        connections // map thread inputs/outputs to process inputs/outputs
                connect_ss_sub_out: port create_subscription_thread.output -> output;
                connect_ss_sub_in: port input -> create_subscription_thread.
                    input;
                connect_ss_mon_out: port create_monitored_item_thread.output
                    -> output;
                connect_ss_mon_in: port input -> create_monitored_item_thread
                    .input;
end OPCUASubscriptionServiceSetProcess.impl;

---- OPC UA CreateSubscription service ----
subprogram CreateSubscription // represents ResponseHeader
    create_subscription(...)
        features
                response_header: out parameter opcua::ResponseHeader.impl; //
                     identifies the ResponseHeader type to be returned
                subscription_id: out parameter opcua::IntegerId;
                revised_publishing_interval: out parameter opcua::Duration;
                revised_lifetime_count: out parameter opcua::Counter;
                revised_max_keep_alive_count: out parameter opcua::Counter;
                requested_publishing_interval: in parameter opcua::Duration;
                requested_lifetime_count: in parameter opcua::Counter;
                requested_max_keep_alive_count: in parameter opcua::Counter;
                max_notifications_per_publish: in parameter opcua::Counter;
                publishing_enabled: in parameter Base_Types::Boolean;
                priority: in parameter Base_Types::Integer; // Integer to be
                    assumed as Integer_8 which is used instead of octet
                    according to the specification
        properties
                Source_Language => (C); // implementation language is C
                Source_Name => "CreateSubscription"; // name of the
                    corresponding C function
                Source_Text => ("opcuaservicesetmethods.c"); //
                    implementation file
end CreateSubscription;

thread CreateSubscriptionThread
        features
                output: out data port opcua::OPCUAData.impl;
                input: in data port opcua::OPCUAData.impl;
        properties
                Dispatch_Protocol => Aperiodic; // as service requests are
                    asynchronous
                Compute_Execution_Time => 10 ms .. 30 ms; // estimation of
                    the time required for generating a proper response to a
                    service request
end CreateSubscriptionThread;

thread implementation CreateSubscriptionThread.impl
        calls
                mycalls: {
                        cs_spg: subprogram CreateSubscription;
                };
end CreateSubscriptionThread.impl;

```

```
---- OPC UA CreateMonitoredItem service ----
subprogram CreateMonitoredItem // represents ResponseHeader
    create_monitored_items(...)
        features
                response_header: out parameter opcua::ResponseHeader.impl; //
                     identifies the ResponseHeader type to be returned
                results: out parameter opcuaddssub::
                    MonitoredItemCreateResults.impl;
                diagnostic_infos: out parameter opcua::DiagnosticInfos.impl;
                subscription_id: in parameter opcua::IntegerId;
                timestamps_to_return: in parameter opcua::TimestampsToReturn;
                items_to_create: in parameter opcuaddssub::
                    MonitoredItemCreateRequests.impl;
        properties
                Source_Language => (C); // implementation language is C
                Source_Name => "CreateMonitoredItem"; // name of the
                    corresponding C function
                Source_Text => ("opcuaservicesetmethods.c"); //
                    implementation file
end CreateMonitoredItem;

thread CreateMonitoredItemThread
        features
                output: out data port opcua::OPCUAData.impl;
                input: in data port opcua::OPCUAData.impl;
        properties
                Dispatch_Protocol => Aperiodic; // as service requests are
                    asynchronous
                Compute_Execution_Time => 10 ms .. 30 ms; // estimation of
                    the time required for generating a proper response to a
                    service request
end CreateMonitoredItemThread;

thread implementation CreateMonitoredItemThread.impl
        calls
                mycalls: {
                        cmi_spg: subprogram CreateMonitoredItem;
                };
end CreateMonitoredItemThread.impl;
```

Listing 4.7: AADL model implementation for the OPC UA Subscription Service Set

Analog to the OPC UA Subscription Service Set AADL model, all Service Set models are specified accordingly and in compliance with the OPC UA/DDS Gateway specification.

4.5 Modeling of DDS Components

As a counterpart of the OPC UA AADL system model, the DDS AADL system model is composed of two different AADL models which each specifies relevant entities of the DDS domain required for the integration of the DDS system and the validation of the

OPC UA/DSS Gateway system model. The DDS system model generally defines all DDS types and entities as data models that are specified by the OPC UA/DDS Gateway specification for the DDS domain [8]. In addition, the DDSApplication system model represents DDS domain participants (including the integrated DDS endpoints used in the OPC UA to DDS Bridges and the DDS to OPC UA Bridges), the DDS domain mapping to OPC UA, the representation of DDS topics in OPC UA and the DDS instance methods model (including the representation of instances and samples in OPC UA). The latter is debated in *Section 4.5.2*. The AADL model for the DDS is illustrated in Fig. 4.5.

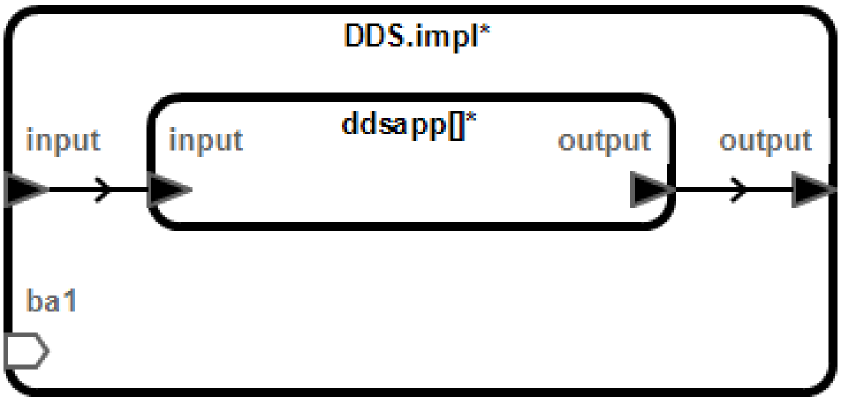

Figure 4.5: Structure diagram of the AADL system model for the DDS

4.5.1 Modeling of DDS types in OPC UA

Analog to the modeling of OPC UA types in DDS 4.5.1, the AADL data modeling of DDS types in OPC UA is exemplary introduced by defining the DDS structure type named ShapeType which is associated to the DDS topic Circle and later on integrated in the AddressSpace of some OPC UA server inside the OPC UA/DDS Gateway AADL system model. The representation of DDS types in OPC UA is used by the DDS to OPC UA Bridges to appropriately map the data types between the domains. The ShapeType DDS type is represented in XML format in Listing 4.8.

```
<types>
        <struct name="ShapeType">
                <member name="color" stringMaxLength="128" type="string" key=
                    "true"/>
                <member name="x" type="int32"/>
                <member name="y" type="int32"/>
                <member name="shapesize" type="int32"/>
        </struct>
</types>
```

Listing 4.8: XML representation of the ShapeType DDS type

The XML format specifies the types clause which defines the description of new DDS types in OPC UA. The struct clause identifies the ShapeType as DDS structure type which is composed of four members called color, x, y and shapesize. Besides the member named color which is a string type, each member represents a 32 bit integer type. The color member type also specifies a maximum length of 128 characters and the keyed type property. The ShapeType DDS type has the following representation in AADL as shown in Listing 4.9.

```
data ShapeTypeDataType
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeDataType;

data implementation ShapeTypeDataType.impl -- data type
        subcomponents
                BrowseName: data String8.impl; -- <StructureTypeName>DataType -> ShapeTypeDataType
                color: data ShapeTypeColor.impl; -- keyed type -> key = true ; stringMaxLength="128"
                x: data ShapeTypeX.impl;
                y: data ShapeTypeY.impl;
                shapesize: data ShapeTypeShapeSize.impl;
end ShapeTypeDataType.impl;

data ShapeTypeVariableType extends BaseDataVariableType
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeVariableType;

data implementation ShapeTypeVariableType.impl -- variable type
        subcomponents
                BrowseName: data String8.impl; -- <StructureTypeName> VariableType -> ShapeTypeVariableType
                DataType: data ShapeTypeDataType.impl; -- shape type data type
end ShapeTypeVariableType.impl;

data ShapeTypeInstance
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeInstance;

data implementation ShapeTypeInstance.impl -- variable/instance
        subcomponents
                BrowseName: data String8.impl; -- Name of the ShapeTypeInstance instance of ShapeTypeDataType
                ValueRank: data UInt32.impl; -- If the Variable represents a Primitive Type, ValueRank shall be set to 0
                color: data ShapeTypeColor.impl;
                x: data ShapeTypeX.impl;
                y: data ShapeTypeY.impl;
                shapesize: data ShapeTypeShapeSize.impl;
```

```
end ShapeTypeInstance.impl;

data ShapeTypeColor -- variable
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeColor;

data implementation ShapeTypeColor.impl
        subcomponents
                BrowseName: data String8.impl; -- ShapeTypeColor
                DataType: data String8.impl; -- stringMaxLength="128"
                ValueRank: data UInt32.impl; -- If the Variable represents a
                    Primitive Type, ValueRank shall be set to 0
end ShapeTypeColor.impl;

data ShapeTypeX -- variable
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeX;

data implementation ShapeTypeX.impl
        subcomponents
                BrowseName: data String8.impl; -- ShapeTypeX
                DataType: data Int32.impl;
                ValueRank: data UInt32.impl; -- If the Variable represents a
                    Primitive Type, ValueRank shall be set to 0
end ShapeTypeX.impl;

data ShapeTypeY -- variable
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeY;

data implementation ShapeTypeY.impl
        subcomponents
                BrowseName: data String8.impl; -- ShapeTypeY
                DataType: data Int32.impl;
                ValueRank: data UInt32.impl; -- If the Variable represents a
                    Primitive Type, ValueRank shall be set to 0
end ShapeTypeY.impl;

data ShapeTypeShapeSize -- variable
        properties
                Data_Model::Data_Representation => Struct;
end ShapeTypeShapeSize;

data implementation ShapeTypeShapeSize.impl
        subcomponents
                BrowseName: data String8.impl; -- ShapeTypeShapeSize
                DataType: data Int32.impl;
                ValueRank: data UInt32.impl; -- If the Variable represents a
                    Primitive Type, ValueRank shall be set to 0
```

```
end ShapeTypeShapeSize.impl;
```

Listing 4.9: AADL representation of the ShapeType DDS type

According to the OPC UA/DDS Gateway specification [8] on mapping DDS types to OPC UA, the ShapeType DDS type first defines a ShapeTypeDataType data model specifying the ShapeType datatype as OPC UA type. Second, a data model representing the ShapeType variable type (ShapeTypeVariableType) and the ShapeType instance (ShapeTypeInstance) was created. Last but not least, a data model for each member of the ShapeType DDS type was created to allow OPC UA to represent instances of the ShapeType.

In comparison with the mapping of DDS enumeration types, bitmask types, union types and collection types, the general approach of modeling more complex DDS types in OPC UA like the ShapeType DDS structure type shown above is the same, thus a detailed explanation of the mapping of all other complex DDS types including sequence types, array types and map types is left out at this point. In order to view those, the entire DDS type to OPC UA type mapping is included in the AADL code of the GitLab project for the OPC UA/DDS Gateway AADL system model which is referenced in the Appendix *Chapter 8*.

Another example demonstrating the mapping of DDS types to OPC UA is shown by the mapping of DDS topics to OPC UA. The mapping of the DDS TopicType also includes a data model for the RegisteredTypeName DDS type which is shown below in Listing 4.10.

```
data TopicType extends dds::BaseObjectType
        properties
                Data_Model::Data_Representation => Struct;
end TopicType;

data implementation TopicType.impl
        subcomponents
                BrowseName: data Base_Types::String; -- TopicType
                RegisteredTypeName: data RegisteredTypeName.impl;
end TopicType.impl;

data RegisteredTypeName
        properties
                Data_Model::Data_Representation => Struct;
end RegisteredTypeName;

data implementation RegisteredTypeName.impl
        subcomponents
                BrowseName: data Base_Types::String; -- RegisteredTypeName
                DataType: data Base_Types::String;
end RegisteredTypeName.impl;
```

Listing 4.10: Representation of DDS topics in OPC UA

4.5.2 Modeling of DDS Instance Methods

Similar to the mapping of OPC UA Service Sets in DDS 4.4.2 which are represented by a hierarchical structure of processes, threads and subprogram component types and component implementations, the AADL model of DDS instance methods in OPC UA was designed using the same approach. Listing 4.11 shows the overall execution model for the DDS instance methods mapping to OPC UA.

```
process DDSAppInstanceProcess -- OPCUAService interface Instance
        features
                output: out data port dds::DDSData.impl;
                input: in data port dds::DDSData.impl;
end DDSAppInstanceProcess;

process implementation DDSAppInstanceProcess.impl
        subcomponents
                register_instance_thread: thread RegisterInstanceThread.impl
                    [];
                unregister_instance_thread: thread UnregisterInstanceThread.
                    impl[];
                dispose_instance_thread: thread DisposeInstanceThread.impl[];
                connections -- map thread inputs/outputs to process inputs/
                    outputs
                connect_reginst_out: port register_instance_thread.output ->
                    output;
                connect_reginst_in: port input -> register_instance_thread.
                    input;
                connect_unreginst_out: port unregister_instance_thread.output
                     -> output;
                connect_unreginst_in: port input ->
                    unregister_instance_thread.input;
                connect_dispinst_out: port dispose_instance_thread.output ->
                    output;
                connect_dispinst_in: port input -> dispose_instance_thread.
                    input;
end DDSAppInstanceProcess.impl;

-- Return Values for StatusCode:
-- Good - The operation was successful.
-- Bad_InvalidArgument - One or more arguments are invalid.
-- Bad_NodeExists - The Node to be created as a consequence of the invocation
     to RegisterInstance already exists.

-- only for topics with a keyed type
subprogram RegisterInstance -- represents StatusCode RegisterInstance(...)
        features
                BrowseName: in parameter Base_Types::String; --
                    RegisterInstance
                -- in <EquivalentType> <key_member_1_name>;
                -- [...in <EquivalentType> <key_member_N_name>;]
                InputArguments: in parameter ddsopcuabridge::Arguments.impl;
                statuscode: out parameter opcua::StatusCode.impl; --
                    identifies the StatusCode type to be returned
        properties
                Source_Language => (C); -- implementation language is C
                Source_Name => "RegisterInstance"; -- name of the corresponding C function
                Source_Text => ("ddsinstancemethods.c"); -- implementation file
end RegisterInstance;

thread RegisterInstanceThread
        features
                output: out data port dds::DDSData.impl;
                input: in data port dds::DDSData.impl;
        properties
                Dispatch_Protocol => Aperiodic; -- as service requests are asynchronous
                Compute_Execution_Time => 10 ms .. 30 ms; -- estimation of the time required for generating a proper response to a service request
end RegisterInstanceThread;

thread implementation RegisterInstanceThread.impl
        calls
                mycalls: {
                        ri_spg : subprogram RegisterInstance;
                };
end RegisterInstanceThread.impl;

-- Return Values for StatusCode:
-- Good - The operation was successful.
-- Bad_InvalidArgument - One or more arguments are invalid.

-- only for topics with a keyed type
subprogram UnregisterInstance -- represents StatusCode UnregisterInstance(...)
        features
                BrowseName: in parameter Base_Types::String; -- UnregisterInstance
                -- in <EquivalentType> <key_member_1_name>;
                -- [...in <EquivalentType> <key_member_N_name>;]
                InputArguments: in parameter ddsopcuabridge::Arguments.impl;
                statuscode: out parameter opcua::StatusCode.impl; -- identifies the StatusCode type to be returned
        properties
                Source_Language => (C); -- implementation language is C
                Source_Name => "UnregisterInstance"; -- name of the corresponding C function
                Source_Text => ("ddsinstancemethods.c"); -- implementation file
end UnregisterInstance;

thread UnregisterInstanceThread
        features
                output: out data port dds::DDSData.impl;
                input: in data port dds::DDSData.impl;
        properties
                Dispatch_Protocol => Aperiodic; -- as service requests are asynchronous
                Compute_Execution_Time => 10 ms .. 30 ms; -- estimation of the time required for generating a proper response to a service request
end UnregisterInstanceThread;

thread implementation UnregisterInstanceThread.impl
        calls
                mycalls: {
                        ui_spg: subprogram UnregisterInstance;
                };
end UnregisterInstanceThread.impl;

-- Return Values for StatusCode:
-- Good - The operation was successful.
-- Bad_InvalidArgument - One or more arguments are invalid.

-- only for topics with a keyed type
subprogram DisposeInstance -- represents StatusCode DisposeInstance(...)
        features
                BrowseName: in parameter Base_Types::String; -- DisposeInstance
                -- in <EquivalentType> <key_member_1_name>;
                -- [...in <EquivalentType> <key_member_N_name>;]
                InputArguments: in parameter ddsopcuabridge::Arguments.impl;
                statuscode: out parameter opcua::StatusCode.impl; -- identifies the StatusCode type to be returned
        properties
                Source_Language => (C); -- implementation language is C
                Source_Name => "DisposeInstance"; -- name of the corresponding C function
                Source_Text => ("ddsinstancemethods.c"); -- implementation file
end DisposeInstance;

thread DisposeInstanceThread
        features
                output: out data port dds::DDSData.impl;
                input: in data port dds::DDSData.impl;
        properties
                Dispatch_Protocol => Aperiodic; -- as service requests  are asynchronous
                Compute_Execution_Time => 10 ms .. 30 ms; -- estimation of the time required for generating a proper response to a service request
end DisposeInstanceThread;

thread implementation DisposeInstanceThread.impl
        calls
                mycalls: {
```

```
                                di_spg: subprogram DisposeInstance;
                      };
end DisposeInstanceThread.impl;
```

Listing 4.11: Representation of DDS instance methods in OPC UA

CHAPTER 5

Dynamic Configurability of the OPC UA/DDS Gateway

Based on the formal AADL system model for the OPC UA/DDS Gateway presented in *Chapter 4*, this chapter describes how to create a dynamically configurable OPC UA/DDS Gateway implementation using the formal AADL system model. For this reason, the code generation process, which creates an equivalent C code representation of the AADL system model using Ocarina is outlined in *Section 5.1.* Furthermore, the problems that occurred during the code generation using Ocarina are discussed in *Section 5.2.* To that end, the approach of creating a dynamically configurable OPC UA/DDS Gateway application is illustrated in *Section 5.3.*

5.1 Code generation process using Ocarina

The C code generation for the AADL model of the OPC UA/DDS Gateway using Ocarina was performed on a Linux Mint 19 x86_64 system architecture. In order to start with the code generation process, the Ocarina build script has to be downloaded from the stated GitHub source [26] [1]. An important note at this point is that the Ocarina build script requires a GNAT/Ada compiler, *gprbuild*, *autoconf* and *automake.* By using the packet management system Advanced Packaging Tool (APT) [27] for instance, all of the dependencies can be installed with the $sudo\ apt - get\ install$ command. After the successful installation of the required tools, the following command installs Ocarina as a fresh installation after cloning the GitHub repository:

$./build\_ocarina.sh\ {-}{-}scenario = fresh{-}install\ {-}{-}prefix = \$PWD/ocarina\_install$
$-t$

[1]The README.md provides detailed information on how the setup of the script works and how to perform a fresh installation

Once the Ocarina build script is executed, the Ocarina repository, the PolyOrb-Hi-Ada/PolyOrb-Hi-C runtime and the aadlib are updated. Afterwards, *make* and *make install* are called to setup and configure Ocarina. Additionally, we specified the command line option $-t$ to run the provided test suite on Ocarina and to verify the installation.

In order to execute the Ocarina code generation script, located in the *ocarina_install/bin* folder, several command-line options and arguments have to be specified to parse the AADL model correctly. As Ocarina includes the PolyOrb-Hi-Ada and the PolyOrb-Hi-C middleware, either Ada or C source code may be generated based on the AADL specification. For this particular case, the following command line string generates the C source code based on the OPC UA/DDS Gateway AADL model for the target platform:

```
./ocarina −aadlv2 −f −p −g polyorb_hi_c −r Complete.impl −o
/git/aadl_opcuaddsgateway/aadl
/git/aadl_opcuaddsgateway/aadl/ddsapp.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsattribute.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddssub.aadl
/git/aadl_opcuaddsgateway/aadl/dds.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsmethod.aadl
/git/aadl_opcuaddsgateway/aadl/overview.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaclient.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsgateway.aadl
/git/aadl_opcuaddsgateway/aadl/opcua.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsview.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaserver.aadl
/git/aadl_opcuaddsgateway/aadl/ddsopcuabridge.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsquery.aadl
/git/aadl_opcuaddsgateway/aadl/opcuaddsbridge.aadl
```

The flag $-aadlv2$ specifies that AADLv2 is used to interpret the AADL specification. The flag $-f$ parses pre-defined non-standard property sets while the flag $-p$ parses and instantiates the AADL models. The option $-g$ specifies the middleware that has to be used for the code generation process. The option $-r$ defines the name of the AADL root system implementation and the flag $-o$ declares the output directory. The arguments provided to the Ocarina script represent all the AADL files necessary to successfully parse the AADL model of the specified AADL root system implementation. If the Ocarina script does not print any output on the command line and returns, this means that the code generation process for the provided AADL model has been finished successfully. Otherwise, the errors, as stated by the command line output of the Ocarina script have to be checked, and the AADL model has to be modified, respectively. For providing a general overview, the basic code generation process using Ocarina is visualised in Fig. 5.1. [2].

[2]For further details on the command line options provided by Ocarina please check: https://ocarina.readthedocs.io/en/latest/usage.html

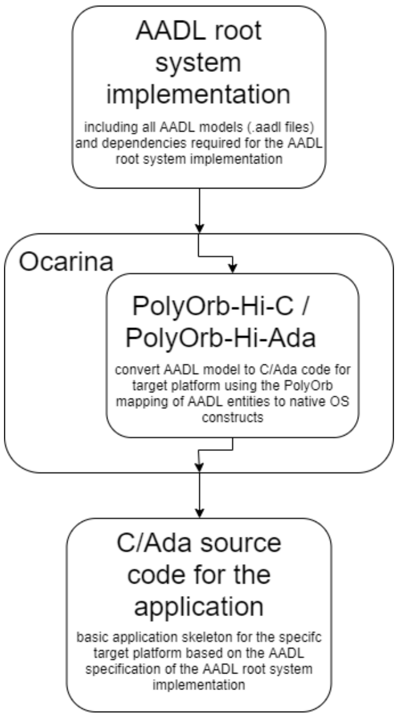
AADL root system implementation
including all AADL models (.aadl files) and dependencies required for the AADL root system implementation
Ocarina
PolyOrb-Hi-C / PolyOrb-Hi-Ada
convert AADL model to C/Ada code for target platform using the PolyOrb mapping of AADL entities to native OS constructs
C/Ada source code for the application
basic application skeleton for the specifc target platform based on the AADL specification of the AADL root system implementation

Figure 5.1: Flow chart illustrating the code generation process using Ocarina

According to the mapping of AADL entities/components, as specified by PolyOrb-Hi-C [28], a target folder with the name of the AADL root system implementation is created. This folder further contains several node folders which each represent a process within the generated distributed application. Each node folder is composed of a Makefile and several C source files specifying all the related PolyOrb-Hi-C components relevant for building the executable/node including subprogram interfaces, types, requests, activities, naming, deployment, marshallers as well as the main and a Doxygen description of the node.

5.2 Experienced Problems

This section states all problems encountered during the setup of Ocarina and the AADL specification of the OPC UA/DDS Gateway model. It also points out the specific issues discovered during the code generation process of the modeled AADL system model for the OPC UA/DDS Gateway.

First off, it was tried to setup Ocarina on a machine featuring Windows 10 Home 64 bit. For this purpose, the Ocarina build script [26] was used, and all the prerequisites as discussed in *Section 5.1* have been installed for Windows accordingly. During the update process of the Ocarina build script, some errors concerning the autoconf and automake configuration occurred though (some modules could not be adequately loaded). It was not possible to solve this issue and required a switch to a Linux Mint 19 64 bit virtual machine. On the Linux VM, it was possible to install Ocarina.

In a first attempt to start the code generation process, the Ocarina plugin for OSATE was installed and configured in OSATE. This plugin should allow starting the code generation process from inside OSATE by selecting an AADL root system implementation from the Outline. Unfortunately, Ocarina did not create a proper C code implementation as expected from the explanation of the documentation. As there were no traceable errors that could explain the malfunctioning of the plugin, the next code generation attempt focused on using Ocarina from the command line interface.

After the first execution of the Ocarina script on the command line by providing the necessary options and arguments in order to parse the AADL model, several errors appeared. Ocarina could not semantically parse some data models, modeled as Structure Type (Data_Representation => Struct) besides being syntactically correct according to the AADL specification. In order to resolve this problem, remodeling the particular data models was the only option. Moreover, Ocarina stated an error regarding the Base_Types::String component type as the maximum length of a string has to be defined. The error has been resolved by modifying the Base_Types package and specifying the Source_Data_Bits property (set to 8) and the Dimension property (set to (128)) which are defined by the Data_Model package for the String component type (represents a 128 bytes bounded string). Furthermore, the AADL specification of how a thread is supposed to call a subprogram using the Compute_Entrypoint property for some particular thread had to be changed. A thread component now specifies the calls clause in its component implementation and explicitly declares the subprogram call by referencing the name of the subprogram component type.

Additionally, Ocarina does not support the modeling of data models that represent union types. The fix for this issue is that these data models also specify a structure type using the Data_Representation property. In order to distinguish between an actual structure type and a union type, the first subcomponent in a union type component implementation has to represent the identifier indicated by an enumeration type.

Ocarina further requires the binding of connections between features of system implementation and process implementation to a dedicated bus component implementation of the particular system model which caused the error that a connection has to be bound to a bus. After defining a designated bus component for the system model and by specifying the Actual_Connection_Binding property to bind each connection to the bus subcomponent implementation, this issue was solved.

The last error is caused by calling make for some arbitrary node after the Ocarina toolchain seemingly successfully generated a C code implementation. Specifically, the error states that some variable declaration is faulty, although Ocarina generated the corresponding C file. Until the end of the thesis, it was not possible to fix this problem.

5.3 A dynamically configurable OPC UA/DDS Gateway application

Based on the problem description stated in *Section 5.2*, this section discusses all the essential steps that are assumed to be necessary in order to create a dynamically configurable OPC UA/DDS Gateway application.

Once the Ocarina code generation completes the C code based on the formal OPC UA/DDS Gateway AADL system model appears in the specified output directory. The output directory which corresponds to the AADL root system implementation represents the root node of a hierarchical structure of nodes as described in *Section 5.1.* In order to run the distributed application, each node may be compiled individually using the provided Makefile for each node or the Makefile in the root directory can be used to compile all nodes at once. Each node executes, after running the dedicated binary executables,

The dynamic configurability of the OPC UA/DDS Gateway is supposed to be guaranteed by the invocation of two steps depending on the intended runtime configuration.

5.3.1 Dynamic Data type compile-time configuration

In the first case, the generated PolyOrb-Hi-C code implementation defines the configuration of the OPC UA/DDS Gateway. For this purpose, the first scenario assumes that only some values concerning a particular set of data types/data models need to change in order to meet the new runtime requirements.

PolyOrb-Hi-C generally creates an application skeleton and declares the data types for each node in the corresponding types.h file. These data types are parameters for some specific subprogram in AADL. At the same time, they are defined as parameters of subprogram corresponding functions in the PolyOrb-Hi-C code (as specified in the subprograms.c/subprograms.h files). Each subprogram or set of subprograms is executed by some dedicated type of thread or also called job. The activity.c file of each node describes the set of threads that are executed by a node and each thread's implementation. Therefore, the initialisation/configuration of the specific data type values is located at the startup routine of each thread. All threads only specify input parameters though, as output parameters do not require initialisation values because they depend on the functional aspects of the subprogram/function. In order to guarantee a high configurability of the application, a DDS Consolidated XML (XSD) to C parser is proposed based on the OPC UA/DDS Gateway specification [8] which translates/maps the XML configuration accordingly for all the specified data types and initialises the corresponding data types of all threads in the activity.c file with the pre-defined values from the XML file. Thereby, the XSD file contains all the required configuration values for the existing type definition in order to enable the intended configuration of the OPC UA/DDS Gateway. After the parser completes, the new configuration is loaded, and the application can be compiled and run again.

5.3.2 Dynamic Data model extension for the compile-time configuration

In the second case, the configuration of the OPC UA/DDS Gateway relies on additional data types necessary to meet the runtime requirements specified by the user. For efficiently achieving this, an additional parser is proposed which receives an XML file as input that corresponds to XSD syntax (.xds files) as stated by the OPC UA/DDS Gateway specification [8]. The XSD file is composed of all the new type definitions required for the intended configuration of the OPC UA/DDS Gateway. The parser then fetches all type definitions, translates/maps the XML syntax of the type definitions into AADL syntax and supplementary writes/appends the type definitions based on the correct modeling approach for the specific data types to the designated AADL models. After the AADL system model contains all the required data models for the new configuration, Ocarina is used to parse the AADL model into PolyOrb-Hi-C code. The remaining steps can be performed analogue to the description in *Section 5.3.1*.

With the concept of having two separate parsing mechanisms, one for adding new type definitions and one for assigning a dedicated compile-time configuration to all existing type definitions and entities specifically, a XSD syntax may be easily parsed and configured for the OPC UA/DDS Gateway application by combining the benefits of both mechanisms as illustrated in Fig. 5.2. Thus, it shall also be possible to provide one XSD file which enables the XSD to AADL parser to add new type definitions/data types to the AADL model. In contrast, the XSD to C parser allows interpreting the remaining XML specification to create a valid compile-time configuration for the application (Ocarina has to be called before the XSD to C parser can operate on the updated version of the application).

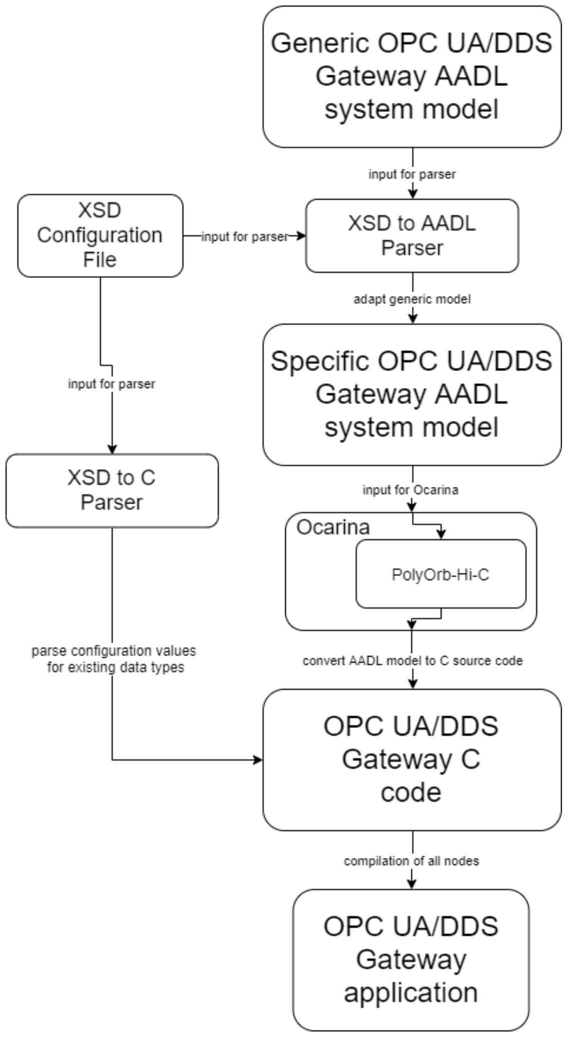
Generic OPC UA/DDS Gateway AADL system model
input for parser
XSD Configuration File
input for parser
XSD to AADL Parser
adapt generic model
input for parser
Specific OPC UA/DDS Gateway AADL system model
XSD to C Parser
input for Ocarina
Ocarina
PolyOrb-Hi-C
parse configuration values for existing data types
convert AADL model to C source code
OPC UA/DDS Gateway C code
compilation of all nodes
OPC UA/DDS Gateway application

Figure 5.2: Flow chart illustrating the process of dynamically configuring the OPC UA/DDS Gateway

CHAPTER 6

Discussion on the dynamically configurable OPC UA/DDS Gateway

Subsequent to the modeling of the OPC UA/DDS Gateway in AADL 4 and the dynamic configurability of the OPC UA/DDS Gateway 5, this chapter outlines and discusses the work that was done in this thesis. First off, the compliance of the implemented OPC UA/DDS Gateway AADL system model with the OPC UA/DDS Gateway specification [8] is discussed 6.1. This section also demonstrates the relevance of the OPC UA/DDS Gateway AADL model and compares our AADL model with the previous work that has been done as stated in the Related Work *Section 2.1*. Finally, in *Section 6.2*, we conclude our discussion by answering the research questions that have been formulated in *Chapter 1*.

6.1 Conformance of the OPC UA/DDS Gateway AADL model with the OPC UA/DDS Gateway specification

According to the OPC UA/DDS Gateway specification, the conformance of an OPC UA/DDS Gateway implementation or model is based on the corresponding building blocks that have been implemented/modeled. These building blocks specify the level of accuracy between an implementation/model and the specification. For this purpose, the second chapter of the OPC UA/DDS Gateway specification [8] states corresponding conformance criteria by defining four individual conformance points which are composed of specific build blocks that have to be considered in order to reason about the compliance of the OPC UA/DDS Gateway AADL model with the specification.

The first two conformance points define the compliance level of the OPC UA to DDS Mapping. Thus, an implementation/model either satisfies the basic or the complete compliance criteria. For the basic compliance, an implementation/model requires the implementation of the OPC UA Type System Mapping and the OPC UA Subscription Model Mapping. In order to guarantee complete compliance, the basic compliance requirements have to be satisfied while the OPC UA Service Sets Mapping additionally has to be implemented. Regarding the provided implementation of the OPC UA/DDS Gateway AADL system model, the basic OPC UA to DDS Mapping compliance requirements are satisfied as the OPC UA Type System Mapping and the OPC UA Subscription Model Mapping are completely modeled and specified by the OPC UA to DDS Bridge AADL model, the OPC UA AADL model and the OPC UA Subscription Service Set AADL model. The complete compliance with the OPC UA to DDS Mapping is achieved through the implementation of the OPC UA Service Sets Mapping which is supplementary modeled and specified by the following AADL models/packages: OPC UA Client, OPC UA View Service Set, OPC UA Query Service Set, OPC UA Method Service Set and OPC UA Attribute Service Set. For this reason, the OPC UA/DDS Gateway AADL model fully conforms with the OPC UA to DDS Mapping.

Analog to the first two conformance points, the third and the fourth conformance point specify the compliance level of the DDS to OPC UA Mapping. In this scenario, the third conformance point targets the basic compliance of the DDS to OPC UA Mapping by the implementation of the DDS Type System Mapping and the DDS Global Data Space Mapping building blocks except for the sub clause: Reading Historical Data From Instance Nodes. In order to satisfy the DDS to OPC UA Mapping basic conformance requirements, the DDS to OPC UA Bridge AADL model and the DDS AADL model completely model and implement the appropriate building blocks. For guaranteeing the complete conformance with the DDS to OPC UA Mapping, the sub clause: Reading Historical Data From Instance Nodes has also been implemented in the DDS AADL model. To that end, the provided implementation of the OPC UA/DDS Gateway AADL model further completely complies with the DDS to OPC UA Mapping.

Based on the satisfaction of the requirements necessary for the complete conformance with the OPC UA to DDS Mapping and the DDS to OPC UA Mapping, our formal AADL model implementation of the OPC UA/DDS Gateway 4 fully conforms with the OPC UA/DDS Gateway specification and provides a formal and complete bi-directional communication model between OPC UA and DDS applications. Contrarily, the AADL system model represents a frame for the concrete implementation though, as further data models may be added for the OPC UA to DDS Mapping or the DDS to OPC UA Mapping by a provided XSD configuration file for certain use cases. In some specific application scenario, the XSD to AADL parser receives the appropriate XSD configuration file as input and converts the XSD specification of the type definitions into corresponding AADL data models in order to generate the specific OPC UA/DDS Gateway AADL model based on the generic OPC UA/DDS Gateway AADL model.

In extension, the provided formal AADL model implementation of the OPC UA/DDS Gateway may also be used in order to create a dynamically configurable OPC UA/DDS Gateway application as comprehensively discussed and visualised in Chapter 5. For this purpose, the AADL model may serve as a complete formal model reference implementation which can be built upon and used to generate a platform specific configurable OPC UA/DDS Gateway application.

In comparison with *Section 2.1*, specifically the hybrid implementation proposed in [13] which defines the mapping of OPC UA data types to DDS data types, our formal AADL model implementation also defines the DDS to OPC UA Mapping and is fully conformant with the OPC UA/DDS Gateway specification respectively. In relation to the concept of a smart gateway middleware connecting the OPC UA and the DDS [9], we do not provide an evaluation of our AADL model compared to the evaluation of the concept for the smart gateway middleware using a Raspberry Pi. Instead, we provide a complete formal model implementation and state the concept on how to create an application based on our model and how to make the application dynamically configurable.

6.2 Discussion of the results regarding the research questions

To that end, we want to discuss and illustrate the results of this thesis by answering the research questions that have been formulated in *Chapter 1.*

Research question one (RQ1) aims towards gaining knowledge on how to create an artefact (OPC UA/DDS Gateway) by using the AADL model language.

RQ 1: How to implement the OPC UA/DDS Gateway in the AADL?

Following the design and creation research strategy, as described in *Chapter 3*, *Chapter 4* presents the entire creation process of the OPC UA/DDS Gateway implementation. Further, it contains the primary concept and the hierarchical structure of the AADL system and component types and implementations, accompanied by insights and learnings about the creation process. A particular focus lies on the used AADL modeling approaches for the OPC UA to DDS Mapping and the DDS to OPC UA Mapping according to the specification [8]. *Chapter 8*, *Section 8.1* provides the link to the GitLab project of the thesis containing the entire AADL code of the OPC UA/DDS Gateway root system model, including all corresponding AADL models. The previous *Section 6.1* evaluated the model according to the specification and added valuable insights. Therefore, the sum of all gained knowledge answers RQ1.

The second research question RQ2 aims towards how the artifact performs in its application domain. In particular, what are the necessary steps to create a dynamically configurable OPC UA/DDS Gateway for several application platforms.

RQ 2: How to dynamically configure the OPC UA/DDS Gateway for several application platforms?

For answering RQ2, *Chapter 5* presents a process to transform the formal AADL model into a specific OPC UA/DDS Gateway instance. The chapter displays the code generation process using Ocarina and experienced problems. By introducing a XSD to AADL parser and a XSD to C parser, it is possible to configure the generic AADL model and the application C source code based on a specific XSD configuration file at compile-time. In the last step, all nodes are compiled into appropriately configured executables for the distributed application. Due to the not solvable problems during the code generation process, it was not possible to evaluate a specific OPC UA/DDS Gateway instance.

In order to sum up the results of this thesis, the discussion of the presented work fulfils the aims of the thesis to model the OPC UA/DDS Gateway using the AADL and to discuss on how the OPC UA/DDS Gateway can be dynamically configured based on the formal AADL model.

CHAPTER 7

Conclusion

In modern automation industry, IIoT [2] represents the primary concept of developing highly sophisticated, scalable and efficient automation systems. These types of systems have to be reliable, interoperable and adaptable in order to satisfy the requirements. As OPC UA and DDS represent common standards used in the domain of automation systems, a complete configurable formal model of the OPC UA/DDS Gateway would satisfy the IIoT requirements regarding interoperability, scalability and reliability while also preserving efficiency. Based on this motivation, this thesis presents the entire AADL model implementation process for the OPC UA/DDS Gateway, the conformity evaluation and the specific learning outcomes. It further contains a procedure on how to configure the formal general model towards an executable specific gateway instance. The work provides knowledge about the used toolchain and subsequently answers two research questions to fulfil the scientific requirements for a bachelor thesis.

Following the stated delimitations, the thesis provides the AADL model for further development instead of an executable binary. Future activities should focus on the encountered code generation issues, how to model particular OPC UA and DDS entities in AADL and the extension of the general model. Moreover, the final gateway artefact should be extensively tested concerning reconfigurability, scalability, safety and security.

CHAPTER 8

Appendix

This chapter provides a link to the GitLab project of the thesis which contains the entire AADL code of the OPC UA/DDS Gateway system model 8.1 as well as additionally gives some supplementary, more in-depth technical background on the modeling language AADL regarding the safety analysis and the error model annex 8.2.

8.1 AADL Code for the OPC UA/DDS Gateway system model

The AADL code for the OPC UA/DDS Gateway system model can be accessed via the following link which leads to the GitLab project of the thesis:

https://git.auto.tuwien.ac.at/dramsauer/aadl_opcuaddsgateway/

8.2 Safety Analysis in AADL

The safety analysis in AADL provides many different analysis methods for which the system model may be analyzed to detect several distinct types of possible design, modeling or future implementation flaws. The difference between these analysis methods is that each considers various issues. For this purpose, AADL provides many analysis methods to verify and guarantee schedulability of processes and threads, expected throughput of processors, latency requirements of a logical or physical connection, correct memory resource utilization, proper error analysis of faulty software and hardware components, reliability requirements and last but not least security requirements of the system. Two specific analysis methods AADL uses to verify the reliability requirements of a system model are the Failure Modes and Effects Analysis (FMEA) and the Fault-Tree Analysis (FTA) for instance.

In order to efficiently and clearly integrate the analysis capabilites provided by AADL into the system model, AADL defines the error model annex. The error model annex specifies how error analysis is correctly done by defining error sources and how the occurrence of errors is correctly handled by relying on error propagations for each relevant component and subsystem individually. For getting a general overview of the error model annex, the model is discussed in *Section 8.2.1* in more detail.

8.2.1 AADL Error Model Annex

In this Section, the general basics of the error model annex are explained as well as an introduction to hazards and hazard modeling is given. First off, this thesis considers version 2.0 of the AADL error model annex which is also referenced as Error Model Version 2.0 (EMV2) in the AADL error modeling framework [29]. The main objective of the error model annex is to assess the dependability of a system and guarantee its compliance with the specified fault tolerance strategies for the entire system architecture. The basic concept of the error model annex builds on the definition of error sources to specify where errors may occur and which types of errors are possible for a certain type of component. Thus, the change of a components state to a faulty state has to be monitored which can be specified by error transitions. These define the change of a components state based on an event. Additionally, the error behaviour of that component has to be considered and specified. For example, if some component is fail-safe or fail-operational or more general if a component is operational or already failed. Once an error occurs, the error needs to be propagated to the component or subsystem of the next higher level (hierarchical model-based design). This mechanism is provided by the error propagation specification and is required to generally identify that an error occurred and for which component that error occurred, so that a reaction to that error from the next higher level component or subsystem is possible (fault interaction of components). Complementary, to contain the affects of an error within certain boundaries or at a specific level of the system hierarchy (not every fault needs to be propagated to the very top-level system model), the error containment mechanism may be used to immediately deal with the error at the current level. Supplementary to the error propagation mechanism, the error path also needs to be specified for each component in the system to mitigate the impact of a fault as each component or subsystem in the system model must know its next higher level component or subsystem to which the error has to be propagated.

For that reason, annex uses error type libraries which already provide default/standard error types but also allow the declaration of an arbitrary number of application specific error types. One error model library may be specified per AADL package. Each error model library consists of multiple error type libraries that each may have a hierarchy of multiple error types and error type sets. An error type indicates the actual type of fault of a component category. An error type set represents a set of error type instances. Error types are unique and provide the benefit of defining specific types of faults related to a component that are actually relevant for the considered component or component group. Error type hierarchies enable the separation of fault categories as one error type hierarchy may correspond to another fault category than another error type hierarchy. For example, one error type hierarchy may be related to value faults while another error type hierarchy may be related to timing faults. In addition, severity and likelihood of individual faults and/or fault categories for one or many components may be specified.

In order to illustrate the integration of the error model annex in an application context of a component, the following example shown by Listing 8.1 demonstrates a device type component named TempSensor1 that represents a temperature sensor and defines several sensor-specific error types and enables error propagation for selective error sources. This example is similar to the example provided by Figure 8 of Section "ERROR MODELING WITH AADL" in [29].

```
device TempSensor1 // type definition of TempSensor1
features
current_temp: out data port; // current temperature reading of TempSensor1
Iso_Variables::current_temp; // specify current_temp as iso variable to
    enable the definition of error type severity and propability for
    current_temp
annex EMV2
{**
error types
ValueError : type; // value returned out of expected bounds
TimingError : type; // value not returned during a valid time interval
LowerBoundValueError : type extends ValueError; // value less than lower
    bound (still value error)
UpperBoundValueError : type extends ValueError; // value greater than upper
    bound (still value error)
end types;
use behavior ErrorModel::TempSensorErrorModel; // specify error behaviour by
    the usage of the external error model called TempSensorErrorModel of the
    ErrorModel package
error propagations
use types ErrorLibrary; // useful if error types are intended to be
    externally defined. Not necessary in this example though, as error types
    are internally defined
current_temp : out propagation {ValueError, TimingError}; // propagate only
    errors of type ValueError and TimingError
flows
flow_temp: error source current_temp {ValueError, TimingError}; // specify
    current_temp as potential error source
properties
EMV2::Occurrence => Iso_Properties::TemperatureSensorValueError applies to
    current_temp.ValueError; // defines the temperature sensor value error
    propability (propability pre-defined in iso properties)
EMV2::Occurrence => Iso_Properties::TemperatureSensorTimingError applies to
    current_temp.TimingError; // defines the temperature sensor timing error
    propability (propability pre-defined in iso properties)
end propagations;
**}
end TempSensor1;

device implementation TempSensor1.impl // implementation of TempSensor1
```

```
end TempSensor1.impl;
```

Listing 8.1: Error types and error propagation of a temperature sensor device named TempSensor1

The hardware device with the functional intent as temperature sensor defines one data output port returning the current temperature reading of the sensor. The current_temp feature is also defined as iso variable to further specify error type probabilities using properties and the "applies to" operator which assigns an iso property to the corresponding error type of current_temp. The error types clause defines four types of errors for the temperature sensor: ValueError, TimingError, LowerBoundValueError and UpperBoundValueError. The later two error types are subtypes of the ValueError type which is indicated by the extends clause used in the definition of the two error types. The use behavior clause imports the externally specified error model and its error behaviour from a declared package which is applied to the component. For internally defining the requested error behaviour, the error behavior clause may be used. Additionally, the composite error behavior clause allows to configure multiple state transitions between the operational and failed state of a component. For instance, if one or more subcomponents fail, the component transitions from the operational state into the failed state as well. The error propagations clause defines the propagation of certain error types for each interface. In the example shown by Listing 8.1, the current_temp feature defines to propagate an error if the error type is either a ValueError or a TimingError. The "out" operator specifies that the error is propagated onwards to another component and not received from another component. Finally, the flows clause specifies the origin of the fault (error source) and/or the forwarding source and destination interface for the fault (error path).

Furthermore, annex enables the specification of component hazards to allow a functional hazard assessment of the system model [30]. Hazards are erroneous states indicating a deviation of the expected behavior of a component. The analysis of hazards and hazard sources is of great importance as the origin of hazards and their impact on the overall system may cause severe safety violations and lead to the malfunction of the system. Thus, the following example illustrated by Listing 8.2 visualizes the application of several hazard definitions for a device type component using the hazard model library.

```
device TempSensor2 // type definition of TempSensor2
features
current_temp: out data port; // current temperature reading of TempSensor2
annex EMV2 // error type model
{**
...
**}
annex Hazard_ErrorModel // hazard model
{**
hazard types HazardTypes // define hazard types
hazards
overheating: hazard{Severity => B}; // overheating of the temperature sensor may lead to heat control or general system failures
outage: hazard{Severity => A}; // outage of the temperature sensor due to some issue may cause severe malfunctions in the system
end types;
hazard behavior HazardBehavior
hazard trigger mechanism // defines how hazards may occur
spark: trigger {Occurrence => poisson 0.01}; // occurrence of a spark with poisson distribution and propability 0.01
cooling_error : trigger {Occurrence => latency 3.0} // occurrence of a cooling malfunction with latency 3.0
transitions // defines the transitions between hazards
overheating - [spark] -> outage; // overheating hazard may lead to an outage hazard of a greater severity level when triggered by a spark
end behavior;
hazard sources
device_malfunction: error state => device_failed; // hazard triggered
hazard propagations
state device_malfunction - [cooling_error] -> overheating; // failed state of the temperature sensor and a cooling failure lead to overheating of the temperature sensor
**}
end TempSensor2;

device implementation TempSensor2.impl // implementation of TempSensor2
end TempSensor2.impl;
```

Listing 8.2: Hazard definition for a temperature sensor device named TempSensor2

In extension to the component type definition of the temperature sensor TempSensor1 in Listing 8.1, TempSensor2 also includes certain types of hazard definitions which is shown by Listing 8.2. For this purpose, annex allows the definition of multiple error models for any type of component separating individual error models to provide a better overview and enable a modular configuration of a components error specification. In this example, TempSensor2 was extended by a hazard model which includes the definition of the hazard types overheating and outage. Both hazard types are also identified by their severity level that is assumed to equal the criticality level of an error. Furthermore, the hazard behavior defines mechanisms of how hazards may be triggered and transitions of how the individual hazard types may affect other hazard types. Additionally, hazard sources and hazard propagations are specified to state the origin of a hazard and which hazard source may lead to another more severe hazard. Analog to the error model library, the hazard model library builds on the concept of the fault, error and failure definitions as a single fault in a subcomponent may affect other components or subsystems leading to a possible malfunction of the overall system if not properly handled. For instance, a fault in the temperature sensor may result in an error in the heat control algorithm of a controller subsystem and further in a malfunction of the controller system.

Once all the necessary safety analysis specifications, definitions and properties are integrated in the components of the system model using the error model annex, the system model may be analysed by performing any type of supported and relevant analysis method for the verification of the system model in any particular application context. In order to get an overview of the supported analysis methods by AADL/OSATE and how certain analysis methods may be performed for the verification of the OPC UA/DDS Gateway system model depends on the toolchain that is used in the modeling process which is further discussed in *Section 8.3*.

8.3 Analysis Methods

In order to verify the integrity of the OPC UA/DDS Gateway system model, several analysis methods supported by OSATE may be applied to the system model which each considers a different aspect of application required to satisfy the safety specification of the OPC UA/DDS Gateway. All of these analysis methods propose an individual approach with its own benefits and drawbacks. Hence, the following subsections describe each analysis method individually.

8.3.1 Fault Hazard Assessment

The Fault Hazard Assessment (FHA) generates a detailed document of the system examination that identifies and classifies all potential failures which occur in a system architecture according to their severity. The FHA creates a report for each identified failure and describes possible impacts and design constraints of the system due to that failure.

8.3.2 Fault-Tree Analysis

The *Fault-Tree Analysis* (FTA) may be performed for one particular type of error and has the objective to specify any cause of that error which may occur in the system design. Thus, the FTA sets up a fault-tree where the root of the tree represents that particular error and its respective leaves represent the causes of that error. The fault-tree is a hierarchical tree that may has two or more levels depending on the type of error, the system model and the application context to be analysed. The FTA analyses each path and therefore each possible origin leading to that particular error (hence the name Fault-Tree Analysis).

8.3.3 Markov Analysis

The Markov Analysis (MA) transforms the system model into a markov chain model which allows the validation and verification of system safety properties using the notation of the markov chain model. An example of such a system safety property may be the failure probability of a component.

8.3.4 Failure Mode and Effects Analysis

The *Failure Mode and Effects Analysis* (FMEA) represents the complementary analysis method to the *Fault-Tree Analysis* (FTA). Contrarily to the FTA which intends to find all possible causes of a particular error, the FMEA analyses the impact of a particular error on other components and the overall system architecture. For that purpose, the FMEA intends to specify the severity of any particular type of error and in which degree that error affects or constrains the system design.

List of Figures

2.1 General architecture of an OPC UA client illustrating all the major components [17]) . . . 7
2.2 General architecture of an OPC UA server illustrating all the major components [17] . . . 8
2.3 General architecture of the DDS system illustrating all the major components. The cloud represents the global data space of a DDS domain . . . 9
2.4 Structure diagram of the AADL system model for the OPC UA to DDS Bridge 13
2.5 Structure diagram of the AADL system model for the DDS to OPC UA Bridge 14

3.1 Research method overview and elements based on [20] . . . 28
3.2 Timeline of the main toolchain usage used for development . . . 29

4.1 Structure diagram of the general architecture of the OPC UA/DDS Gateway AADL system model . . . 32
4.2 Structure diagram of the AADL system model for the OPC UA to DDS Bridge 33
4.3 Structure diagram of the AADL system model for the DDS to OPC UA Bridge 33
4.4 Structure diagram of the AADL system model for the OPC UA . . . 34
4.5 Structure diagram of the AADL system model for the DDS . . . 40

5.1 Flow chart illustrating the code generation process using Ocarina . . . 51
5.2 Flow chart illustrating the process of dynamically configuring the OPC UA/DDS Gateway . . . 55

Acronyms

AADL	Architecture Analysis & Design Language. 2, 13
APT	Advanced Packaging Tool. 47
DCPS	Data-Centric Publish/Subscribe. 9
DCPS PIM	Data-Centric Publish-Subscribe Platform Independent Model. 10
DDS	Data Distribution Service. 1
EMV2	Error Model Version 2.0. 62
FHA	Fault Hazard Assessment. 66
FMEA	Failure Modes and Effects Analysis. 61
FTA	Fault-Tree Analysis. 61
IDE	Integrated Development Environment. 26
IIoT	Industrial Internet of Things. 1, 10, 59
IO	Input/Output. 22
MA	Markov Analysis. 66
OMG	Object Management Group, Inc.. 1
OPC UA	Open Portable Communication Unified Architecture. 1
PLC	Programmable Logic Controller. 1
QoS	Quality of Service. 9
RPC	Remote Procedure Call. 10

SCADA Supervisory Control and Data Acquisition. 1

UML Unified Modeling Language. 2, 13

XSD DDS Consolidated XML. 51, 52

Bibliography

[1] D. Kiel, C. Arnold, and K.-I. Voigt, "The influence of the Industrial Internet of Things on business models of established manufacturing companies – A business level perspective," *Technovation*, vol. 68, no. C, pp. 4–19, 2017.

[2] H. Lasi, P. Fettke, H. G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business and Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, 2014.

[3] Harp, Derek R and Gregory-Brown, Bengt, "IT / OT Convergence Bridging the Divide," *NexDefense*, p. 23, 2015.

[4] T. J. Williams, "The purdue enterprise reference architecture," in *Proceedings of the JSPE/IFIP TC5/WG5.3 Workshop on the Design of Information Infrastructure Systems for Manufacturing*, DIISM '93, (NLD), p. 43–64, North-Holland Publishing Co., 1993.

[5] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2347–2376, Fourthquarter 2015.

[6] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture.* Springer Science \& Business Media, 2009.

[7] S. A. Boyer, *Scada: Supervisory Control And Data Acquisition.* Research Triangle Park, NC, USA: International Society of Automation, 4th ed., 2009.

[8] I. O. Object Management Group, *OPC UA/DDS Gateway*, February 2019.

[9] R. Endeley, T. Fleming, N. Jin, G. Fehringer, and S. Cammish, "A smart gateway enabling opc ua and dds interoperability," *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, August 2019.

[10] S. R. a. C. M. U. . S. E. I. Jérôme Hugues, "Open aadl." Website: http://www.openaadl.org/, 2009. Online; accessed 21 April 2020.

[11] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, “Towards a formal specification for an aadl behavioural subset using the lnt language,” *International Journal of Business and Systems Research, 2020*, vol. Vol.14 No.2, pp. 162–190, March 2020.

[12] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, “A formal approach to aadl model-based software engineering,” *International Journal on Software Tools for Technology Transfer (2020)*, vol. 22, pp. 219–247, March 2019.

[13] J. Pfrommer, S. Grüner, and F. Palm, “Hybrid opc ua and dds: Combining architectural styles for the industrial internet,” *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2016.

[14] C. M. U. . S. E. Institute, “Osate.” Website: https://osate.org/, 2016 - 2020. Online; accessed 29 June 2020.

[15] T. ParisTech, “Ocarina.” Website: https://ocarina.readthedocs.io/en/latest/, 2003 - 2009. Online; accessed 29 June 2020.

[16] nationales Arbeitsgremium K 931 „Systemaspekte“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik, *OPC unified architecture – Part 1: Overview and concepts (IEC/TR 62541-1:2010)*, July 2011.

[17] I. O. Object Management Group, *OPC Unified Architecture – Part 1: Overview and Concepts*, November 2017.

[18] I. O. Object Management Group, *Data Distribution Service (DDS)*, April 2015.

[19] P. Angelo Corsaro, “The data distribution service tutorial.” Website: https://www.danskebank.com/en-uk/ir/Documents/2015/Q4/PresentationQ42015-Press.pdf, June 2015. Online; accessed 16 April 2020.

[20] B. J. Oates, *Researching Information Systems and Computing.* SAGE Publications, Ltd., 2012 ed., 2005.

[21] S. T. March and G. F. Smith, “Design and natural science research on information technology,” *Decision Support Systems*, vol. 15, no. 4, pp. 251–266, 1995.

[22] P. Checkland, “Soft Systems Methodology: A Thirty Year Retrospective,” *Systems Research and Behavioral Science*, vol. 17, pp. S11–S58, 2000.

[23] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design Science in Information Systems Research,” *MIS Quarterly*, vol. 28, pp. 75–105, 3 2004.

[24] J. Hughes and T. Wood-Harper, “Systems development as a research act,” *Journal of Information Technology*, vol. 14, no. 1, pp. 83–94, 1999.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual.* Addison Wesley Longman, Inc., 1999.

[26] B. Zalila and J. Hugues, “Ocarina build script.” Website: https://github.com/OpenAADL/ocarina-build, 2020. Online; accessed 30 June 2020.

[27] ubuntuusers.de, “Advanced packaging tool.” Website: https://wiki.ubuntuusers.de/APT/, 2004 - 2020. Online; accessed 30 June 2020.

[28] T. ParisTech, “Polyorb-hi/c.” Website: https://ocarina.readthedocs.io/en/latest/polyorb-hi-c.html#generating-code-from-an-aadl-model, 2003 - 2009. Online; accessed 30 June 2020.

[29] B. Larson, J. Hatcliff, K. Fowler, and J. Delange, “Illustrating the aadl error modeling annex (v. 2) using a simple safety-critical medical device,” November 2013.

[30] X. Wei, Y. Dong, M. Yang, N. HU, and H. YE, “Hazard analysis for aadl model,” August 2014.