Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Code generation from on-board software models conforming to the On-board Software Reference Architecture (OSRA) using DLR software technologies

Raghuraj Tarikere Phaniraja Setty

Course of Study: Information Technology (INFOTECH)

Examiner & Supervisor: Steffen Becker (Prof. Dr.-Ing.)

Commenced: October 2, 2017

Completed: April 2, 2018

Abstract

European Space Agency (ESA) and its industrial partners have come up with the On-board Software Reference Architecture (OSRA) with the aim of favoring the adoption of a software reference architecture across their software supply chain. The center of that strategy involves a component model called the Space Component Model (SCM) and the software development process that builds on it. The SCM aims to model application software as a set of independent software components which interact with each other via clearly defined interfaces with certain guarantees. The SCM is present as an Eclipse Modeling Framework (EMF) based Ecore meta model and it comes with a graphical editor called the OSRA SCM Model editor. Although the SCM provides information about how components interact with each other through the provided or required services, it does not provide an implementation of those services. The work presented here in this Master thesis aims at implementing a back-end code generator for OSRA, supporting the general vision that in the future, an application developer would create and configure components for his/her on-board applications and capture the desired component interactions in an SCM model instance. He/she can then generate code skeletons for the model, i.e., all the concurrency behavior, data exchange, type conversion, etc. are automatically handled by the code generator. As a result, the developer can only concentrate on implementing the functional code of each on-board software component, which in-turn results in shorter development cycles and high cost-efficiency. The code generator uses Tasking Framework as a well-formed platform and bases the generated code on it. The Tasking Framework is a portable framework for data flow and event driven cooperative multitasking which is written in a safe subset of C++. It is developed by the group 'On-board Software Systems' of the German Aerospace Center (DLR) department of Software for Space Systems and Interactive Visualization.

Contents

List of Figures

List of Tables

List of Acronyms

AOCS Attitude and Orbit Control System

API Application Programming Interface

ASSERT Automated proof-based System and Software Engineering for Real-Time systems

ATON Autonomous Terrain-based Optical Navigation

AUTOSAR Automotive Open System Architecture

BI Bound Interval

BIRD Bi-spectral Infrared Detection

BSP Board Support Package

CBSE Component-Based Software Engineering

CHESS Composition with Guarantees for High-integrity Embedded Software Components Assembly

CORDET Component Oriented Development Techniques

COTS Commercial-off-the-shelf

DLR Deutsches Zentrum für Luft- und Raumfahrt

DOMENG Framework for Domain Engineering

DSL Domain-Specific language

EMF Eclipse Modeling Framework

EMOF Essential Meta Object Facility

ESA European Space Agency

Eu:CROPIS Euglena Combined Regenerative Organic food Production In Space

FDIR Fault Detection Isolation and Recovery

FIFO First-In First-Out

LEOP Launch and Early Orbit Phase

MAIUS Matterwave Intrferometer in Microgravity

M & C Monitor and Control

MDE Model-Driven Engineering

MIAT Minimum Inter-Arrival Time

OBC-NG On-Board Computer - Next Generation

OBSW On-board software

OSRA On-board Software Reference Architecture

PECA Plan Establish Collect Analyze

PI Provided Interface

PIM Platform Independent Model

POSIX Portable Operating System Interface

PSM Platform Specific Model

PUS Packet Utilization Standard

RCM Ravenscar Computational Model

RI Required Interface

RODOS Realtime Onboard Dependable Operating System

RTEMS Real-Time Executive for Multiprocessor Systems

RTK Real-Time Kernel

RTOS Real-Time Operating System

RUP Rational Unified Process

SAM Schedulability Analysis Model

SAVOIR Space Avionics Open Interface Architecture

SCM Space Component Model

TSP Time Space Partitioning

UML Unified Modeling Langauge

V & V Verification and Validation

VSL Value Specification Language

WCET Worst-Case Execution Time

List of Listings

Chapter 1

Introduction

1.1. Motivation

European space industry has entered an economic era in which the funding availed to future space missions are not expected to grow significantly. At the same time, future missions are expected to achieve more and more challenging scientific goals with upcoming seasons of capped budgets for earth observation, scientific studies and space exploration [31]. This leads to a situation where the activities like mission analysis and system engineering will play a bigger role in the overall economy of the project, with a proportional increase of the time and cost invested on them [31]. The implication of this is that the realization activities, and software development among them are pushed forward in the project schedule and compressed [1]. Also, the complexity of the software product is foreseen to increase significantly to keep pace with rising mission needs, while the cost of the software development is expected to fit in the same or perhaps even decreased budget envelope. This situation is therefore calling for a rise in the cost effectiveness of the software development, thus ultimately increasing the "value" of the software product delivered with a given budget.

On-board software for satellites can be classified as high-integrity real-time software and the realization of the functional contents which add "value" to the product, is subject to stringent requirements at both process and product level in dimensions such as: time and space predictability, safety, dependability, security [31]. Also the software product is subject to extensive verification and validation steps to ascertain its quality [1]. In order to achieve all of this at reduced effort and a constant overall budget and at an acceptable level of quality, the concept of reusable software architecture plays a crucial role. In this context, a software architecture expresses an architectural framework that hosts the functional contents, architectural assumptions that and methodological principles

that majorly contribute to the attainment of the desired quality of the software product [31].

Parallely in the automotive domain, innovative vehicle functions are leading to a continual increase in the complexity of the vehicle architecture. At the same time, requirements are also sometimes contradictory, for example, supporting driver assistance systems in critical driving manoeuvers while also improving fuel economy and also conforming to the environmental standards [14]. Additional challenges include deeper integration of the infotainment and communication with the immediate vehicle environment and with online services. In order to continue to meet these requirements in the future, a new technological approach is required for the ECU software architecture [14].

More insights into the concepts of software architecture and a software reference architecture are given in the subsequent chapters

The initiatives in coming up with a software architecture in both space and automotive industries adopt the approaches based on the Component-Based Software Engineering (CBSE) and Model-Driven Engineering (MDE) which are in recent times gaining huge industrial acceptance in the domain of embedded real-time systems. This is not surprise at all since these two development paradigms promise important advantages such as better and more disciplined software design and increased reuse potential for the former; greater abstraction level and powerful automation capabilities for the latter [6][31]. Many domain specific initiatives have shown that the the higher level of abstraction in the design process facilitated by the MDE allows addressing the non-functional concerns earlier in the development, thereby enabling proactive analysis, maturation and consolidation of the software design [30]. Moreover, the automation capabilities of the MDE infrastructure may ease the generation of lower-level design artifacts and ease the generation of source code products. First steps towards such an automation is the main motivation of this Master thesis.

Thesis Structure

Die Arbeit ist in folgender Weise gegliedert:

Kapitel ?? – ??: Hier werden werden die Grundlagen dieser Arbeit beschrieben.

Kapitel 8 – Results and Conclusions fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

Chapter 2

The On-board Software Reference Architecture (OSRA)

2.1. Introduction

2.1.1. Background

Space industry has recognized for the past decade the need to raise the level of standardization in the avionics system in order to increase the efficiency and reduce cost and schedule in the development [1]. The implementation of such a vision is expected to provide benefits for all the stake-holders in the space community [1]:

Customer Agencies Significant reduction in the project development cost and schedule and the risk involved in software development.

System Integrators Increased competition among stake-holders to deliver at lower price and maintain shorter time-to-market as a result of multi-supplier option.

Supplier Industry Benefits from diversified customer bases and the supplied building blocks would be compatible with software architectures from the software primes such as Thales Alenia Space and Astrium Satellites (EADS Astrium).

Similar initiatives have already been taken across various industries and eg., Automotive Open System Architecture (AUTOSAR) for the automotive industry is worthy mentioning [16]. Space can benefit from these examples, by studies related to how these or similar initiatives were successfully conducted and how they faired. Although the business model is different in the automotive and the space sectors, AUTOSAR demonstrates that the need for standardization is the key irrespective of sectors and is actually driven by the need of the industry to become more competitive [40].

Space primes and on-board software companies have made significant progress and have implemented and/or are implementing principle of reuse on the basis of their internal software reference architectures and building blocks. However, for this standardization to provide maximum benefits, it has to be tackled at the European level rather than at a company level [1].

European Space Agency (ESA) through its two parallel activities, namely Component Oriented Development Techniques (CORDET) and Framework for Domain Engineering (DOMENG) [39], which aimed at increasing the software reuse in on-board software have confirmed that interface standardization allows to efficiently compose the software on the basis of existing and mature building blocks.

To refer to all the ongoing initiatives and to provide a platform for technical discussions related to the vision of avionics development through maximizing reuse and standardization, a Space Avionics Open Interface Architecture (SAVOIR) Advisory Group (SAVOIR Advisory Group) was created. SAVOIR Advisory Group decided to spawn a specific subgroup for on-board software reference architectures called SAVOIR Fair Architecture and Interface Reference Elaboration (SAVOIR FAIRE) working group. On-board Software Reference Architecture (OSRA) is the result of the R&D activities of this group [1].

The OSRA is designed to be a single, common and agreed framework for the definition of the On-board software (OBSW) of the future European Space Agency (ESA) missions [1]. It is based on solid scientific foundations and accompanied by development methodology and architectural practices that fit the domain. A single software system would thus be an "instantiation" of the reference architecture to specific mission needs [31][1].

2.2. Need for software reference architecture

2.2.1. Motivation

According to the ISO/IEC standard ISO 42010 [18], the software architecture is defined as:

"*The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*"

A software architecture is the key to create "good quality" software because it promotes architectural best practices and contributes to the quality of the software. A bad architecture hinders the fulfillment of functional, behavioral, non-functional and life-cycle requirements [31].

According to the "Rational Unified Process (RUP)" [20], the software reference architecture can be defined as:

"*A predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects*"

A software reference architecture prescribes the form of concrete software architectures for a set of systems for which it is developed. So, a reference architecture is a form of "generic" software architecture which prescribes the founding principles, the underlying methodology and the architectural practices that are recognized by the domain stakeholders as the best solution to the construction of a certain class of software systems [31][36].

Elevating a software architecture to a software reference architecture permits to gather and re-use lessons learned and architectural best practices, give new projects a consolidated running start and promote a product line approach [1].

A generic software reference architecture is made up of two main parts [1]:

Software architectural concepts These address the pure software architectural related issues.

Architectural building blocks and interfaces These are related to functional aspects and the corresponding interface definitions which express functions derived from the analysis of the functional chains of the core on-board software domain.

Figure 2.1.: Parts of a software reference architecture

Source: [1]

As mentioned in the previous section, in order to increase the efficiency and cost-effectiveness in the development process of on-board avionics and to incorporate more number of functionalities in the on-board software, the overall objective of space industry would be to standardize the avionics systems and therefore the on-board software.

A building block approach is one of the ways to tackle this problem. In this approach, the on-board software is implemented from a set of pre-developed and compatible

building blocks, plus specific adaptations and "missionisation" according to specific mission requirements [1]. The target missions are the core ESA missions, i.e. high reliability and availability spacecraft driven systems (eg. operational missions, science missions).

The "right" building blocks need to be produced and supplied by the suppliers to any system integrator and to achieve this, reference architectures need to be defined.

A software building block, generally [1]:

- Has a clear, well defined, specified, documented function and open external interfaces for the purpose of interaction.
- Meets defined performance, operation and other requirements.
- Is self-contained so that they can be used at higher-integration levels eg. board, equipment, subsystem.
- Has a quality level that can be assessed.
- Is applicable in well defined physical and hardware environment.
- Is worth developing as they are going to be used in bulk of ESA missions.
- Is designed for reuse in different projects, by different users under different environments.
- Can be made available off-the-shelf, ready for deployment under different conditions.

Separation of the application aspects from the general-purpose data processing aspects is the key to generic/reusable software architectures [31]. The lower layers of the architectures usually handle the implementation of communication, real time capabilities etc. and the higher level layers usually deal with the application aspects. However there have to be ways to annotate the application building blocks (ABB) with sufficient information regarding requirements related to communication, real-time, dependability etc., so that the platform building blocks (PBB) can provide a suitable complete implementation. Development of interface specifications with reference architectures as a basis, allows the implementation of the famous AUTOSAR concept: "*Cooperate on standards, compete on implementation*"[14].

The OBSW life-cycle needs to be consistent with the system life-cycle, which features the definition of functional increments in system development [1]. Hence, OBSW must in particular:

- Allow for faster software development.

- Be compatible to a late definition or changes of some of its requirements.
- Cope with various system integration strategies.

2.2.2. User needs

The COrDeT study, with the slogan "Faster, Later, Software", represented a summary of the above programmatic stakes for the on-board-software life-cycle [39][1]. These stakes are included and defined as the user needs [1][31] for the development of OSRA:

Shorter software development time Need for faster software development in the context of a shorter schedule. The Figure 2.2 depicts the reduction in the schedule for software development in the future projects.

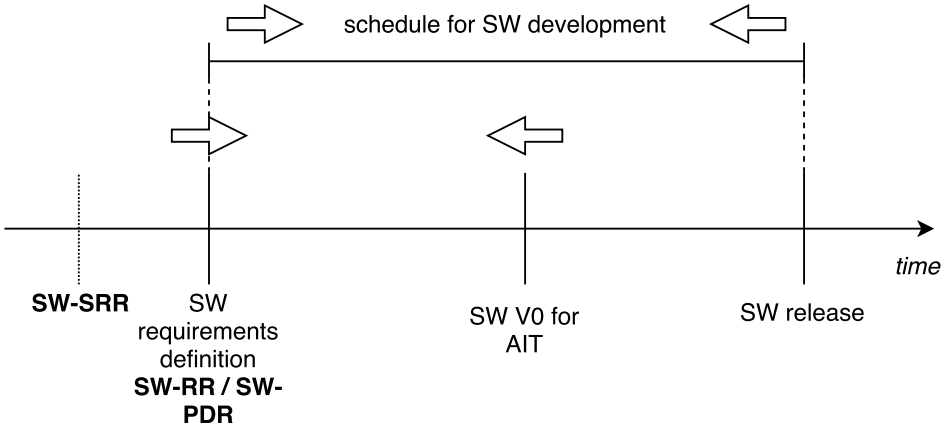


Figure 2.2.: Reduction in the schedule for software development

Source: [1]

Reduce recurring costs Identification and reduction of recurring costs by providing the same set of functions eg., device drivers, real-time operating systems, communication services, etc.

Quality of the product Need for high quality software (timing predictability, dependability, etc.) and the quality must be at least the same as one of OBSW developed with current approaches.

Increase cost-efficiency Increase in the "value" of the software product that is developed for a given amount of budget.

Reduce Verification and Validation effort The new development approaches shall foster the reduction of effort for Verification and Validation (V & V), which is one of the main contributor to the cost of software development.

Mitigate the impact of late requirement definition or change

Support for various system integration strategies It should be possible to do preliminary software releases which allow early system integration efforts.

Simplification and harmonization of FDIR A simplification and hopefully, harmonization of the Fault Detection Isolation and Recovery (FDIR) approach is advocated.

Optimize flight maintenance There should be provision for changing the OBSW during flight maintenance and coordination of strategies to perform it.

Industrial policy support Enable multi-team software development, so that subcontracting to the non-primes is possible, while still being in-charge of the integration.

Role of software suppliers Increase competence of supplier and foster competition among them.

Dissemination activities System engineers should be exposed to the core principles of the process.

Future needs Future needs such as integration of functions of different criticality and security levels, use of Time and Space Partitioning (TSP), support to multi-core processors, need to be subjected to evaluation and their impact on software reference architecture need to be monitored.

2.2.3. High level requirements

The user needs are translated into a set of high-level requirements for OSRA [1][31].

Software reuse The architecture shall be designed in such a way that the reuse of the functional aspects should be independent of the reuse of the non-functional aspects, reuse of the unit, integration and validation tests are made possible.

Separation of concerns Separation of concerns is one of the cornerstone principles of OSRA and it deals with separating different aspects of the software design, in particular the functional and non-functional concerns. Separation of concerns helps to reuse functional concerns independently from non-functional concerns, which increases the software reuse.

Reuse of V&V tests The chosen architectural approach should also promote the reuse of Verification and Validation tests that were performed on the software and not just the software itself. The aim is to maximize the reuse of the tests written for the functional part of the component software.

HW/SW Independence Software should be developed independent from the hardware features. It is necessary to separate parts of the software that interact directly with the hardware, into separate modules and make them accessible through defined interfaces. In this way, as long as the interface does not change, the software is isolated from the changes in the hardware-dependent layers.

Component based approach The whole software should be designed as a composition of components that are reusable in nature. The architecture shall respect preservation of properties of individual building blocks, once they are integrated into the architecture and it should be possible to calculate the system's property as a function of components' individual properties. The former is called composability and the latter is called compositionality [30]. Section 2.3.2.1 in Chapter 3 explains this approach in more detail.

Software observability The software architecture should provide means to observe the software specific parts and extract current and past status of the software using the services specified by its operational scenarios.

Software analysability The design process and methodology used for the reference architecture shall support the verification of functional and non-functional properties at design time.

Property preservation The non-functional properties should be considered as constraints on the system as they specify the "frame" in which the system is expected to behave. These properties have to be preserved or enforced so that these properties are not only used for the analysis of the software model, but also find their way through to the final system at run-time. Adequate mechanisms should be provided to handle the enforcement of the properties and also mechanisms to handle reactions to violation of these properties.

Integration of software building blocks The architecture should allow the combination of coherent building blocks.

Support for variability factors The architecture shall include design features allowing isolating the variability foreseen in the domain of reuse.

Late incorporation of modification in the software The architecture should be immune to late modification of the software in the software life-cycle. System integration almost always finds some system problems and it is the responsibility of the software to contain these problems and implement new requirements. The architecture to which the software is conformal to, should be able to handle these late modifications in the software.

Provision of mechanisms for FDIR The requirements for FDIR, are consolidated often late in the life cycle and the software architecture must accommodate for it.

Software update at run-time The reference architecture should allow update to single software components as well as their bindings without having to reboot the entire on-board computer as it is a risk for the system and reduces the mission availability/up-time.

2.3. The Software Architectural Concept

2.3.1. Fitting Model-Driven Engineering

MDE is a novel trend for software development in the space domain, but has been successfully applied to enterprise computing [30]. The validation-intensive real-time high-integrity systems such as on-board software systems make the adoption of the MDE considerably more arduous. Positive experiences on the application of MDE to the design of these kind of systems do exist and it can be found in the 'Composition with Guarantees for High-integrity Embedded Software Components Assembly (CHESS): space case study' [33] and the 'ESA: reference Earth Observation case study' [33].

In MDE, the principal design artifact is a model, which is an abstract representation of the system under development, which encompasses systems and software architecture. Each model conforms with a metamodel, which describes the syntax of entities that populate the models, as well as their relationships and the constraints in place between them. The metamodel constrains the design space of the MDE infrastructure [13].

COrDeT (Component Oriented Development Techniques) study aimed at investigating various techniques in fields such as software product line engineering, model driven engineering and component orientation [39]. Based on this study, the concept of overall software reference architecture, in the development of OSRA, is considered to be made up of [36][1]:

Component Model A component model is the basis for designing the software as a composition of individually verifiable and reusable software units [29].

Computational Model A computational model is used to relate to the design entities of the component model, their non-functional needs for concurrency, time and space, to a framework consisting of analysis techniques, in general, to a set of schedulability analysis equations, which help to judge formally, whether the description of the architecture is statically analyzable [3].

A Programming Model A programming model is used to ensure that the implementation of the design entities obey the semantics and the assumptions of the analysis and the attributes used as input to it [35].

A conforming Execution Platform An execution platform helps to preserve at runtime, the properties asserted by the static analysis, and is able to react to possible violations of them.

These become the key ingredients for the very foundation of the MDE design methodology focused on the principle of correctness by construction and property preservation, which are high level requirements respectively. Figure 2.3 gives a pictorial representation of it.

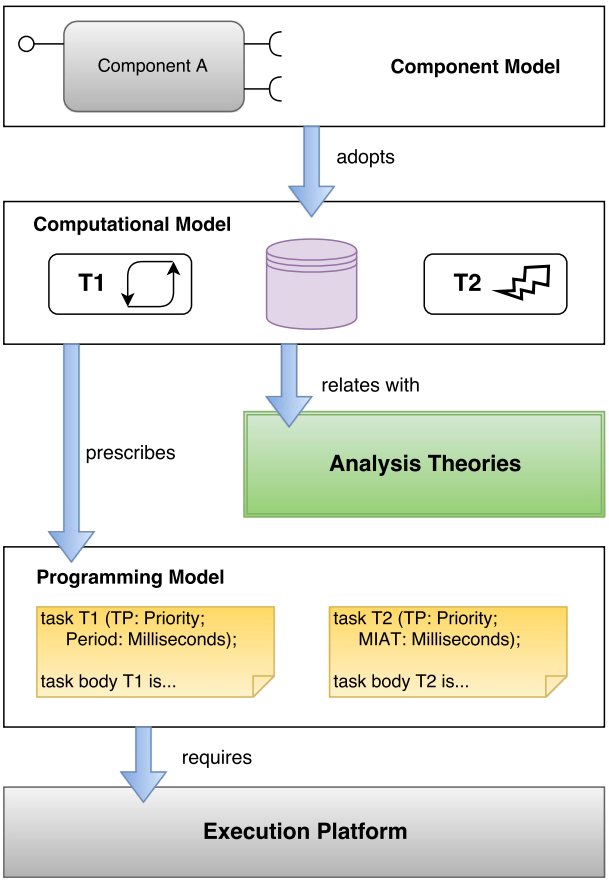


Figure 2.3.: The four constituents of the software reference architecture

Source: [31]

2.3.2. Component Model

CBSE is a software methodology centered on the systematic re-use of software by realizing the software as an assembly of units of composition called components [6]. The adoption of Component Based Software Engineering (CBSE) in the context of

high-integrity real-time systems in general and in space domain in particular is not so popular as in the other main-stream domains like enterprise computing because of strong verification and validation requirements imposed in the space domain and the presence of non-functional dimensions [1].

The principles of Component Based Software Engineering (CBSE) when combined with the principles below can be used to build OBSW as an assembly of components [1]:

- Principle of separation of concerns, by allocation of concerns to three distinct software entities: the component (which is a design entity), the container and the connector (which are entities used in implementation only and which do not appear in the design space).
- Possibility of verification of properties related to composability and compositionality [30].

The execution platform defined in the software architecture then provides the services to the components, container and the connectors. Finally, the entire software is deployed on the physical architecture (Computational units, equipment, and the network interconnections between them).

2.3.2.1. **Founding principles of choice**

This section describes the founding principles of choice of the component model:

Correctness by construction E.W. Dijkstra in his ACM Turing lecture in 1972 suggested that the program construction should be done after a valid proof of correctness of construction has been developed [33]. Two decades later, a software development approach called Correctness by Construction (C-by-C) was proposed which advocated the detection and removal of errors at early stages, which led to safer, cheaper and more reliable software [33][31]. The Correctness by Construction practice follows:

- To give a solid reasoning on the correctness of the document or code, it is necessary to use formal and precise tools and notations for their development and verification.
- Defining things only once so as to avoid contradictions and repetitions.
- Designing the software that is easy to verify e.g. by using safer language subsets or using appropriate coding styles and software design patterns.

In OSRA and the component model developed along with it, the Correctness by Construction principle is changed to be applicable to a CBSE approach based on Model-driven Engineering (MDE) [33] wherein:

- The components can be designed.
- The products designed by the design environment can be verified and analyzed by the design environment.
- The lower level artifacts can be automatically generated and the software production can be automated to the maximum extent.

Separation of concerns Separation of concerns was first advocated by Dijkstra [33] and it helps to separate the aspects of software design and implementation. The OSRA and its associated component model promotes separation of concerns[33][31]:

- The components are restricted to hold the functional code only. The non-functional requirements which has effects on the run-time behavior e.g. tasking, synchronization and timing are dealt by the component infrastructure which is external to the component and which realizes the functional code. The component infrastructure mainly consists of containers, connectors and their run-time support.
- A specific annotation language is specified which is used to define the non-functional requirements and these are annotated on the components realizing the functional code.

  By this, model transformations that automatically produce the containers and connectors that serve the non-functional requirements, enable the execution of the schedulability analysis directly on the model of components. This makes the implementation of the non-functional concerns fully compliant with its specification [3].
- A code generator (whose development is the prime concern of this Master thesis) operates in the back-end of the component model, builds all of the component infrastructure that embeds the user components, their assemblies and the component services that help satisfy the non-functional properties [35].

Inculcating the principle of separation of concerns in the development process has two major benefits [33]:

- It increases the reuse potential of the components, which is an important high level requirement described in the previous section [1]. The reuse potential of the component is increased because the same component can now be used under different non-functional requirements (as per the instantiations of the component infrastructure).
- It helps in the generation of vast amount of complex and delicate infrastructural code which takes care of realizing the non-functional requirements at run-time. This increases the readability, traceability and maintainability of the infrastructural code.

Composition When composability and compositionality can be assured by static analysis, guaranteed through implementation, actively preserved at run-time, the goal of composition with guarantees as discussed by Vardanega can be achieved [33]. This is also one of the high level requirements defined in the section before.

Composability is guaranteed when the properties of individual components are preserved on component composition, on deployment on target and on execution. The components, as mentioned before, implement only the functional code, most part of which is sequential only and they do not have to worry about the non-functional semantics. The components behave like black-boxes and showcase to the external world only the provided and required interfaces. Other components or infrastructural components are expected to communicate through these defined interfaces only. Hence, when components are composed with each other with matching required and provided interfaces, the functional composability is guaranteed which is necessary but not sufficient.

The non-functional requirements/constraints are annotated on the components (specifically the component interfaces) and they are realized by the container which encapsulates the respective component [1][29]. The provided interface determines the semantics of the invocation and adds to the functional capabilities provided by the component. These semantics must match with the execution semantics described by the computational model, to which the component model is attached. An example for composable property is shown in Figure 2.4. In this case, the number of threads and protected objects generated per component, which entirely depends on the extra functional notations to the component interface should be invariant across component composition for the composable property to hold true.

The computational model chosen should help extend composability to the non-functional constraints e.g. concurrency and the ones related to real-time and make it possible to get a compositional view of how execution occurs at the system level. Compositionality is said to be achieved when the properties of the system as a

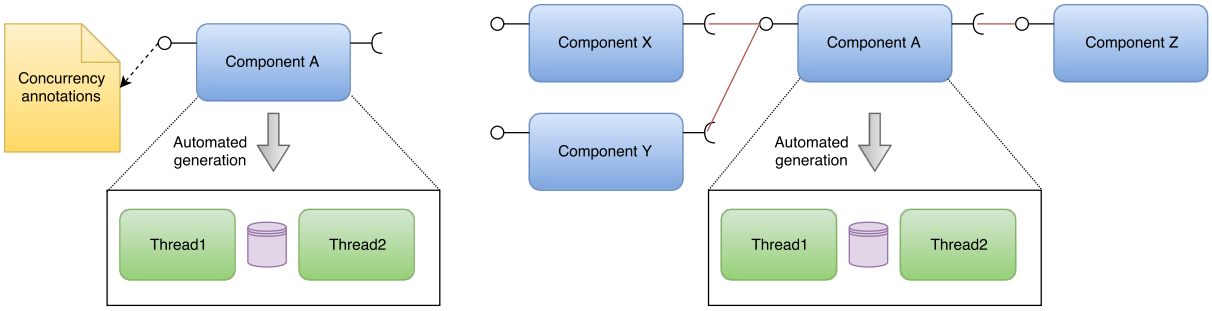


Figure 2.4.: Example for composability

Source: [31]

whole is a function of the properties of the constituting components. Finally, the binding of the computational model to the component should allow the execution semantics of the components with non-functional descriptors to be completely understood. An example for compositionality in shown in Figure 2.5. In this case, it should be possible to calculate the overall latency for the delivery of an output (i.e., the worst-case response time of the end-to-end chain of activities) from individual latencies of different components for the property of compositionality to hold true.

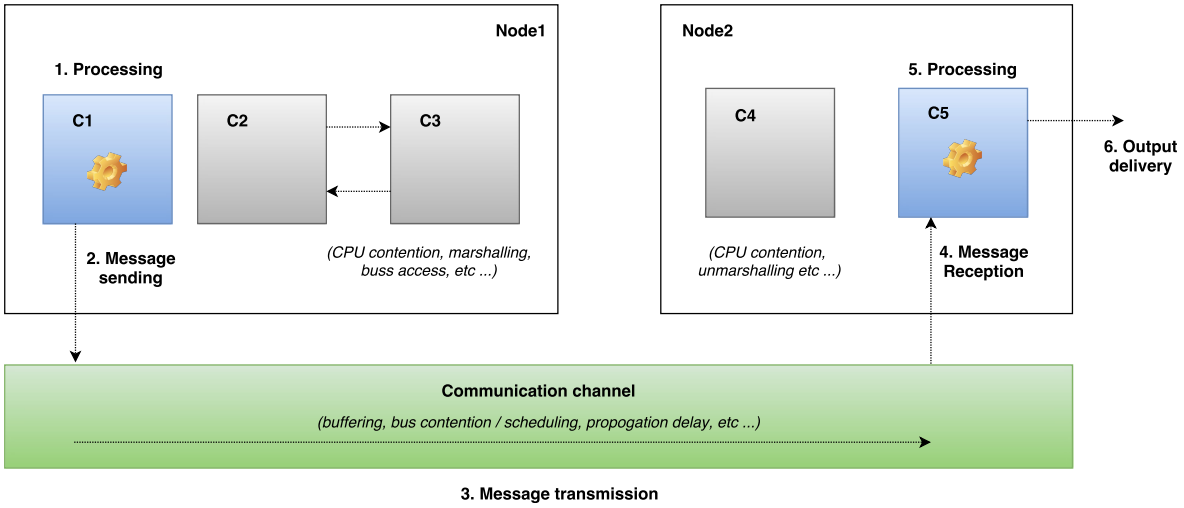


Figure 2.5.: Example for compositionality

Source: [31]

In OSRA, the first and second needs can be met by having correct representations of the non-functional attributes in the component interfaces and the third need is taken care of by the generation of proper code artifacts, which is the main concern of this Master thesis.

2.3.2.2. **Software entities**

The following section describes more about components, containers and the connectors.

Component Chaudron and Crnkovic describe that a Component model defines standards for properties that individual components must satisfy and the methods and possibly ways to compose components [33].

A component provides a set of services and exposes them to the external world as a "provided interface". The service which is needed from other components or the environment in general are declared in a "required interface". A particular component connects to other components in order to satisfy the needs of its required interfaces. An event based communication system is also possible between components and a component can register to an "event service" to get notified about events emitted by other components.

Non functional attributes are added to the component interfaces as discussed before in the previous section on separation of concerns.

The adoption of hierarchical decomposition of components can be an effective way of defining components instead of defining a containment relationships. A child component can be developed to any component which would delegate and subsume the relationships between the interfaces of the child component and its parent. But the drawback is that various non-functional dimensions applicable to the space domain complicate the picture and hence is hierarchical decomposition of components not allowed at the current stage of development [31].

Container The container is a software entity that wraps around the component, which is directly responsible for realizing the non-functional properties. The relation between the component and the container is a famous software design pattern called the "inversion of control" [33][12]. All in all, the reusable code (the container), controls the execution of the problem-specific code (the component).

The container exposes the same provided and required interfaces as that of the component and is able to support the component's execution with the desired, relevant non functional concerns attached to the component interfaces [30]. The container also intercepts the calls made by the component to the other components/services requested from the target platform and transparently forward them to the container of the target component/target platform pseudo component (A pseudo component is a kind of component which is used for interaction purposes only). The former principle is called interface "promotion" and the latter is called the interface "subsumption" [30]. The container and the component interact with

each other according to the inversion of control design pattern, but the binding between components are still defined at software initialization time.

Connector The connector is a software entity responsible for the interaction between the components (actually between the containers that wrap around them). Connectors assist in implementing separation of concerns as the concerns of interaction is separated from the functional concerns. Components are thus void of code related to interactions with other components, however the the component model requires that the user specifies the interaction style in the component interfaces.

The component can be specified independently of the components it eventually binds to, the cardinality of the communication and the location of the other components it connects to, thanks to the principle of separation of concerns.

No complex connectors are necessary in this Master thesis because, a simple linux based PC is chosen as a target system for component deployment and this greatly reduces the variety of connectors needed. Connectors necessary for function/procedure calls (which are usually straight-forward) are sufficient in this Master thesis. One of the major reasons, to go for a simple system is because this Master thesis does not deal with the hardware design or hardware modeling of the on-board software systems.

Figure 2.6 shows a connector mediating a connection between components A and B. The figure also shows components A and B being enveloped by their containers respectively. The containers would be responsible for the realization of the non-functional properties of the respective components they envelop.

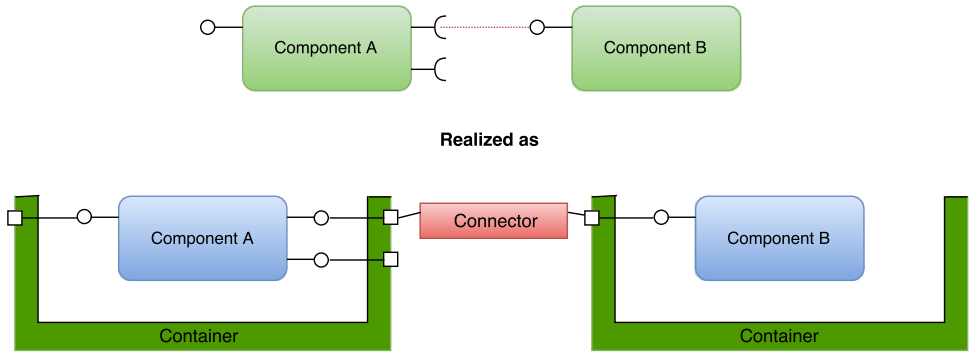


Figure 2.6.: Components, containers and a connector connecting them

Source: [32]

2.3.3. Computational model

Using a computational model is necessary as per the Space Software engineering standard (ECSS-E-ST-440C) standard [1]. A dynamic software architecture is described according to an analyzable computational model which infers that the model development is fully consistent with that which underpins the mathematical equations which are used to predict the schedulability behavior of the system [3]. The computational model is more concerned about entities that belong to the implementation model (eg. tasks, protected objects and semaphores). A more abstract level description of these entities should be provided so that [1]:

- Pollution of the user-models with entities that are more primitive and are of interest to the lower levels of abstraction, is avoided. This is in line with the principle of separation of concerns, which is one of the high-level requirements.
- The abstract representations represent the entities and their semantics faithfully.
- Correct transformation of the information set by the designer in the higher-level representation to entities recognized by the computational model is ensured. This is in line with the principle of property preservation, which is also one of the high level requirements.

2.3.4. Programming model and the execution platform

The execution platform is a part of the software architecture providing all the necessary means for the implementation of a component and the computational model. It comprises of the middleware, the Real-Time Operating System (RTOS)/Real-Time Kernel (RTK), communication drivers and the Board Support Package (BSP) for a given hardware platform. The services provided by the execution platform can be categorized into four different types [1]:

Services for containers These services are meant to be used by the containers e.g. Tasking primitives, synchronization primitives, primitives related to time and timers.

Services for connectors These services are intended to be used by the connectors and it consists of actual communication means between components, ways to handle physical distribution across processing units, libraries used for translating data codes.

Services to components These services are supposed to be used by the components which implement the functional constraints. Typical services include: provision of on-board time for time-stamps, context management and data recovery. Access to these services are intercepted by the container wrapped around the component (refer Section 2.3.2.2).

Services to implement "abstract components" These services include Packet Utilization Standard (PUS) monitoring, OBCPs, hardware representation etc.

It is important to note that different implementation of containers and connectors are necessary for each execution platform of interest.

The programming model realizes a given computational model and together with a conforming execution platform, it is possible to achieve the following goals [31]:

- Ensure that the implementation fully conforms with the semantics prescribed by the computational model and those assumed by the analysis.
- Ensure that the contracts stipulated between the components are respected at run-time. This is in-line with the principle of property preservation which is one of the high level requirements.

The achievement of those two goals would then warrant that the system represented for the analysis purposes is a faithful representation of its implementation and the results of the analysis performed on the system model would be a valid prediction of the system at run-time [31].

The programming model, which is the subject of this Master thesis, is realized by adopting Tasking framework as a computational model whose concurrency semantics would conform to the analysis model.

Chapter 3

Overall component-based software development process

3.1. Introduction

In this chapter the design and implementation steps for the component-based software engineering (CBSE) approach are elaborated. The software design process involves two main actors: the software architect who is responsible for the entire software and provides support at system-level to the customer, and the software supplier who is responsible for the development of part of the software [33]. The parts of the software supplied by the software suppliers are then integrated in the final integration step.

Most of the activities described below come under the responsibility of the software architect, but as soon as the component is defined, it can undergo a detailed design and code implementation. This may indicate some shortcomings and flaws in the design of the component, which might trigger a re-design, re-negotiation of the component definition. This often leads to an iterative/incremental development process [3]. Detailed design and implementation of components are usually done by the software developers or it may be subcontracted to third party software suppliers.

3.2. Design entities and design steps

There are two kind of entities which are defined in OSRA: Design-level entities which are explicitly specified in the design space and require the skills of the user to use them, real-time architecture entities which are not explicitly represented in the design space, instead they are automatically generated by the code-generation engines. The

automatic generation of containers and connectors are possible only upon the knowledge of the computation model and execution platform that are going to be adopted [1][33]. As already mentioned in the previous chapter, this master thesis considers Tasking Framework as the computational model and a normal linux based PC as the execution platform.

The following entities belong to the design space: Data types, events, interfaces, component types, component implementations, component instances, component bindings and the entities required for the description of the hardware topology and platforms. The following entities belong to the real-time architecture: containers and connectors.

The development process is clearly divided into different steps [33][31][1]:

Step 1: Definition of data types and events Data types are the basic entities in the approach and they can be primitive types, enumerations, ranged or constrained types, arrays or composite types (like structs in C or record types in Ada). An event is used in the publish-subscribe communication paradigm and it is an asynchronous message passing scheme.

Step 2: Definition of interfaces A set of operations with one or more already typed parameters, each with a direction (`in, in out, out`) are grouped together to form an interface. The interface can also hold a set of interface attributes of an already defined data type. The interface attributes can have read-only or read-write accesses. From the list of interface attributes, set of `getter` and `setter` operations can be generated for the attribute access, in particular `getter` operations for attributes with `read` only access and `getter` and `setter` operations for attributes with `read-write` access.

Figure 3.1 depicts three data types and an event. Interfaces `AOCS_IF` and `THR_IF` implement only operations while interface `GYR_IF` comprises one read-only attribute.

Step 3: Definition of component types Component types form the basis of a reusable software asset [33]. The software architect defines the component type to provide the specification of the functions that the component of the respective type would implement. The component types are independent of each other and they can consist of:

- One or more provided interfaces, which list the services that the component of the respective type would provide.
- One or more required interfaces, which list the functional services that the component of the respective type would require in order to function correctly according to the functional specifications.

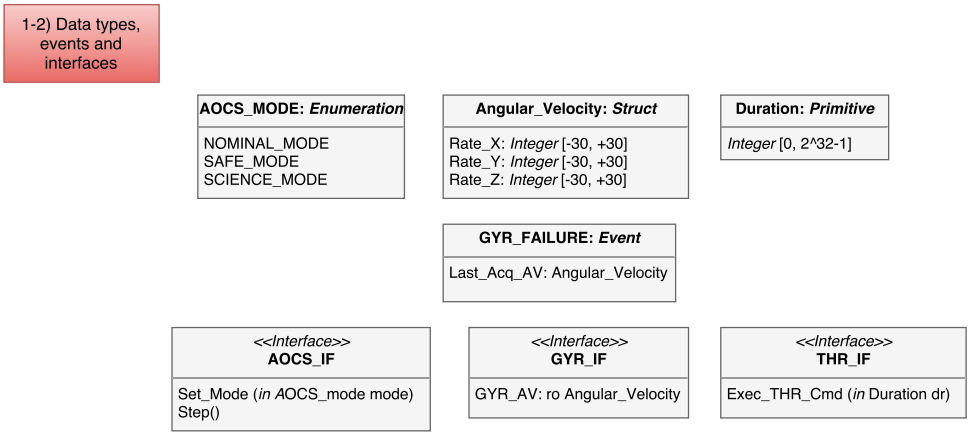


Figure 3.1.: Data types, events and interfaces

Source: [33]

- A set of component type attributes of already defined data types that are local to the component and cannot be accessed from outside.
- Event emitter/receiver ports to raise or receive events.

In order to specify the provided and required interfaces, the component type references the interfaces that were defined in Step 2. This helps in straight forward matching of the required and provided interfaces.

Figure 3.2 depicts a component type `AOCS`. This component type provides interface `AOCS_IF` and requires interfaces `GYR_IF` and `THR_IF`. It also raises events of type `GYR_FAILURE`.

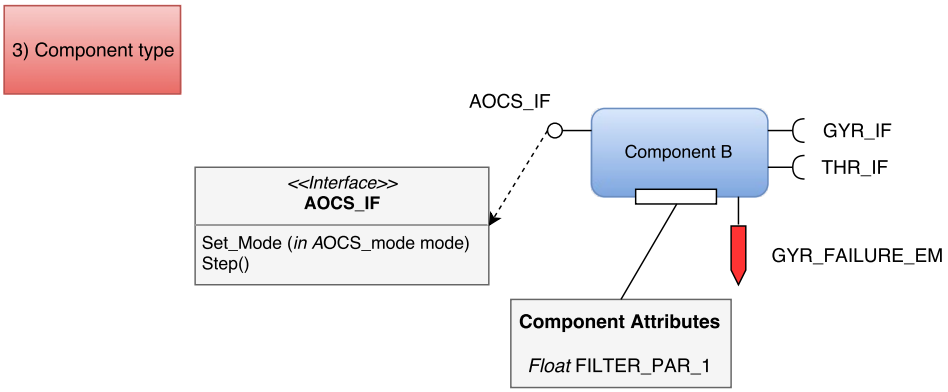


Figure 3.2.: Component type

Source: [33]

Step 4: Definition of component implementations The software architect can create and refine a component implementation from the component type. The component implementation contains the functional code in the form of source code that implements all the services that the component is supposed to provide. It acts as a black box and only its external interfaces are only that matter. It is also a subcontracting unit to the software supplier.

A component type can have more than one implementation and all of these implementations contain only pure sequential code i.e. it is void of any tasking or timing constructs. Implementations can be developed in multiple languages such as Ada, C, C++ etc.

The component implementation should also provide constructs to store the attributes exposed through its provided interfaces and its component type. Technical budgets such as Worst-Case Execution Time (WCET) for a particular operation, maximum memory foot-print for component implementation, maximum number of calls to a certain operation on a required interface, can be placed on the entire component or on the operations and the implementation of the component shall respect this budget. Despite a sequential nature of the code, a component implementation may set specific non-functional constraints to preserve the functional correctness of its behavior. Component implementation is thus a particularly attractive unit to be subcontracted to a third-party software supplier because the software architect can define components, attach technical budgets to it and delegate the implementation to he software suppliers. The software suppliers might add additional operations to the component implementation as and when necessary for the implementation [33].

Figure 3.3 depicts one of the many possible component implementations for the component type `AOCS`.

Step 5: Definition of component instances A component instance is an instance of a component implementation. It is a deployment unit which is subjected to allocation on a processing unit and it is an entity on which the non-functional properties are specified. Specifically, the non-functional properties are attached, as in Figure 3.4, to the provided interface side of the component, as they are the expression of a property or a provision of the component instance.

Step 6: Definition of component bindings Component bindings, as the name suggests, are the connections between one required interface of a component and the provided interface of another component. These bindings are set at design time and is subjected to static type matching to ensure that correct required and provided interfaces are connected to one another. This can be done by asserting the compatibility of the two interfaces (by type system or by inspection of the

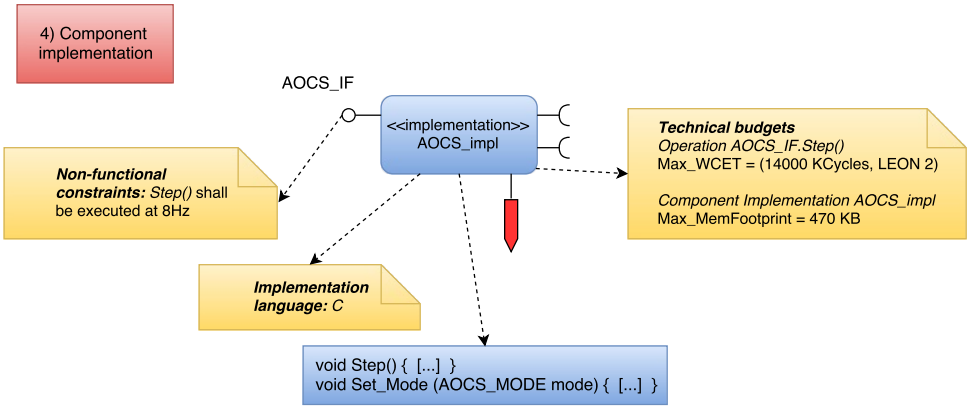


Figure 3.3.: Component Implementation

Source: [33]

signature of their operations). If the binding is legal then whenever a call is made to an operation in the required interface, the call is dispatched to the correct operation in the bound provided interface. The signature of the calling operation in the Required Interface (RI) and the called operation in the Provided Interface (PI) are different and the connector, connecting these two interfaces, is in charge of performing this step. A tool support (possibly a back-end code generator) should initiate the configuration of the connector to perform this kind of binding.

It is also possible in this step to define bindings between an event emitter port of one component and an event receiver port of another component as shown in Figure 3.4.

Step 7: Specification of non-functional attributes After component instances and component bindings have been defined, the software architect can add non-functional attributes to the services of the provided interfaces.

In this step, the software architect specifies the timing and the synchronization attributes [33]. At first, the concurrency kind of the operation is established, and they can be `synchronous` or `asynchronous` operations. In case of a `synchronous` operation, it is executed in the flow of control of the caller and in case of an `asynchronous` operation, the operation is executed by a dedicated flow of control on the side of the callee.

A `synchronous` operation is said to be `protected` if it needs to be protected from data races in case of concurrent calls. The operation is said to be `unprotected` if it is free from such risks. In case of a `asynchronous` operation type, the architect can choose one of the following release patterns for the operation:

Periodic operation The execution platform executes the operation at fixed periods with a dedicated flow of control.

Sporadic operation Two subsequent execution requests are separated by a minimum timespan called the Minimum Inter-Arrival Time (MIAT). The execution platform and the infrastructural code should guarantee this MIAT separation between two subsequent calls to the operation and the component implementer does not have to worry about it.

Bursty operation Only particular number of activations of an operation is allowed in a bounded interval of time. Again, as in the case for sporadic operation, the execution platform and the infrastructure code guarantees this and the component implementer does not have to worry about it.

For all the operations which have concurrency kind set as `asynchronous`, the software architect must provide the worst case execution time (WCET) of the operation. A preliminary value of WCET is initially provided based on previous use of operations in other projects (if any) and they can be refined with bounds at later stages after performing a timing analysis for a given target platform.

Figure 3.4 depicts the component bindings between the required and provided interfaces of the `AOCS` and `Mode_Manager` component instances. It also depicts the non-functional properties which are specified on the services provided by the provided interfaces.

Step 8: Definition of physical architecture The hardware topology provides a description of the system hardware limited to the aspects related to communication, analysis and code generation. It also provides a model-level description of the relevant hardware of the system. In the hardware topology, following elements are described:

- Processing units that have a general-purpose processing capability
- Avionics Equipment/Instruments/Remote terminals
- The interconnection between the elements mentioned above
- A representation of the ground segment/other satellites (eg. Formation flying) to state the connection between the satellite and ground segment or other space segments.

For the specification of these elements, following attributes are used:

Processor frequency This is used for processors to re-scale WCET values expressed in processor cycles in Step 6.

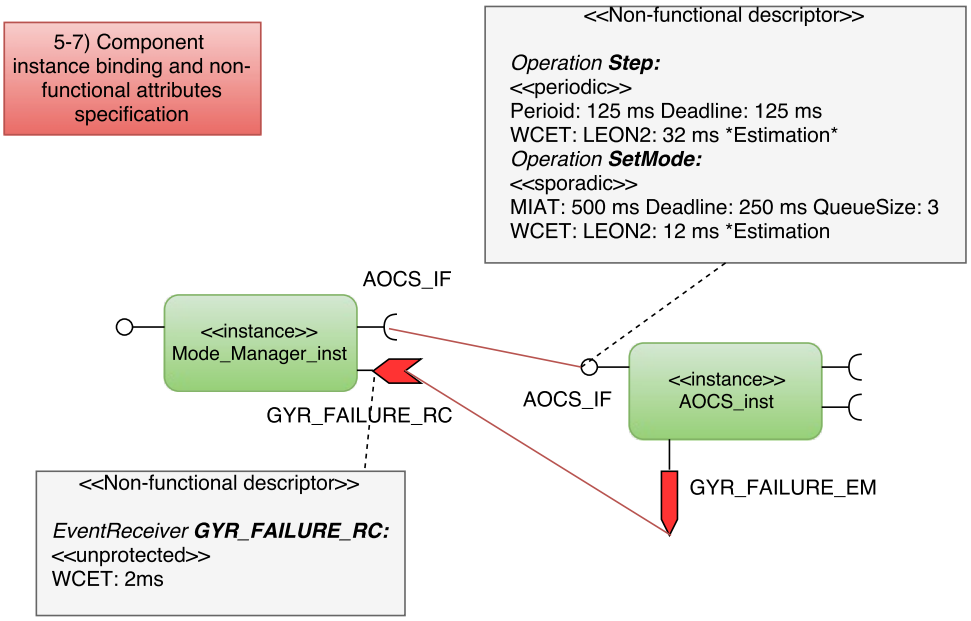


Figure 3.4.: Component instance, component bindings and decoration with non-functional attributes

Source: [33]

Bandwidth This is used for buses and point-to-point links and it indicates maximum blocking time due to non-preemptability of the lower priority message transmission (for whatever reason), minimum and maximum size of packets, minimum and maximum propagation delay, maximum time that the bus arbiter/driver needs to prepare and send a message on the physical channel and maximum time for the message to reach the receiver.

Step 9: Component instances and component bindings deployment In this step, the component instances are allocated on the processing units defined in the hardware topology in Step 8. In majority of the cases, it is straight-forward to allocate the bindings between the components because they are deployed on the same processing units [33]. In other cases, they need to be specifically allocated.

Step 10: Model-based analysis The system model developed within the software reference architecture is subjected to schedulability analysis to determine whether the timing requirements set in the interfaces can be met.

From the user model which is a Platform Independent Model (PIM), a Schedulability Analysis Model (SAM), which is a Platform Specific Model (PSM) is created. This model is subjected to analysis and the results of the analysis is available for the software architect as a read-only result.

The analysis transformation chain requires a model representation of the generated containers and connectors to be defined in the SAM for an accurate analysis [33].

Please note that, step 10 is not of concern in this Master thesis as this Master thesis deals only with automatic generation of containers and connectors and hence an accurate model based schedulability analysis is outside the scope. It is assumed in this Master thesis, that the user models successfully pass the Model based schedulability analysis and hence are subjected directly to the model-code transformation. The actual flow is as depicted in Figure 3.5. Also, Steps 8-9 are not of concern in this Master thesis, as it deals with hardware modeling and they are again outside the current scope of this Master thesis. However, these steps were mentioned for the sake of clarity and continuity.

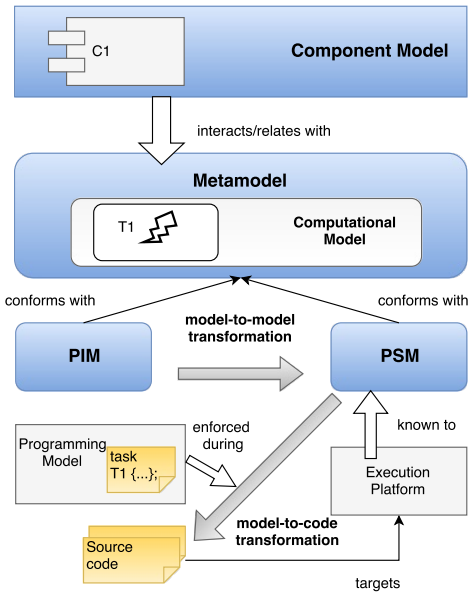


Figure 3.5.: Generation of SAM and model-code transformation

Source: [30]

Step 11: Generation of containers and connectors This step is one of the main focus points of this Master thesis as mentioned before. Containers and connectors are generated and they specify:

- The structure of each container in terms of the required and provided interfaces of the enclosed component that they delegate and subsume.
- The structure of each connector

The non-functional attributes, component instances deployment and component connectors deployment play a major role in determining the creation of connectors

and containers and how component instances and their operations are allocated to them.

Concurrency can be achieved by encapsulating sequential procedures into tasks which reside in containers and the protection from concurrent accesses can be provided by attaching them concurrency control structures. All of this can be achieved without modifying he sequential code and simply by following the use relations among the components.

In order for the OBSW to interact with the external world, sensors and actuators need to be provided. These hardware entities are represented as pseudo components (A pseudo-component indicates that a component is for interaction purposes only) and software capability is attached to these components at the component instance level.

Figure 3.6 depicts the automatic generation of containers and connectors for the components `AOCS` and `Mode_Manager`. It also shows how their component instances are allocated to them.

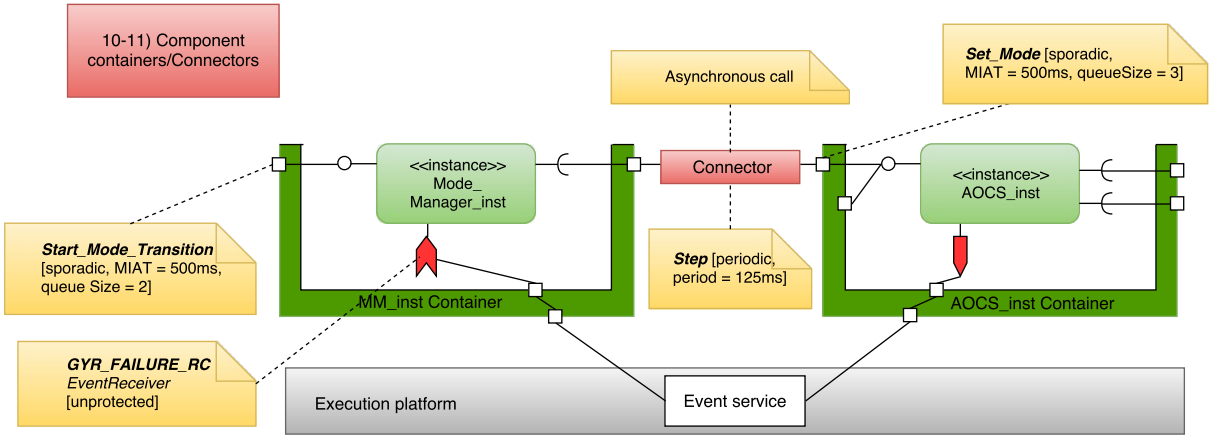


Figure 3.6.: Automated generation of containers and connectors

Source: [33]

3.3. Design flow and design views

When the component model is defined, it also defines implicitly a design flow as shown in Figure 3.7, that needs to be followed, to be able to create an OBSW that meet all of its user needs and high level requirements [1][31][33]. The design flow is as explained in the previous section.

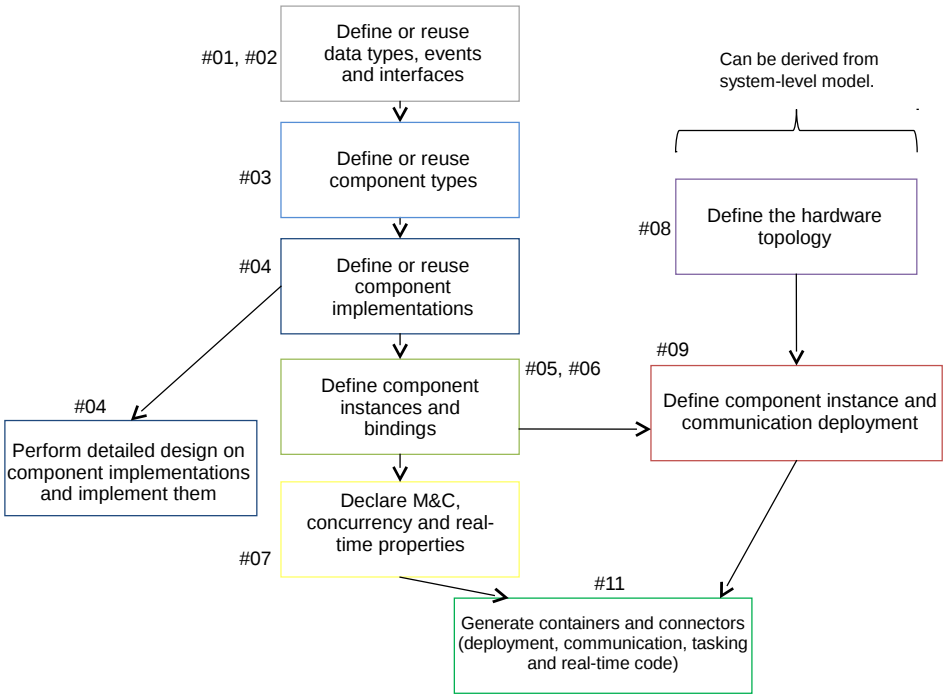


Figure 3.7.: The overall design flow

The Obeo Designer Framework [27] provides a concept called 'Viewpoint' and using this concept, the design views are implemented [33]. One of the advantages of the design views is to promote or enforce a certain design flow [33]. The component model is accompanied by the following design views:

Data view This view is for the description of data types and events.

Component view For definition of interfaces, components and the binding between them to fulfill their required needs.

Hardware view For the specification of the hardware and the network topology.

Deployment view For the allocation of components to computational nodes.

Non-functional view In this view, the non-functional attributes are attached to the functional description of components.

Space-specific view In this view, the services related to the commandability and observability of the spacecraft are specified.

3.4. Language units of the OSRA

The modeling language provided to the software architect to model the OBSW is divided for the ease of construction of the OBSW models into a set of language units [32]. Each language unit consists of closely related metamodel entities. The language units are grouped into separate meta-models for the sake of re-use as shown in Figure 3.8 [32]. OSRA Component model is composed of the following language units:

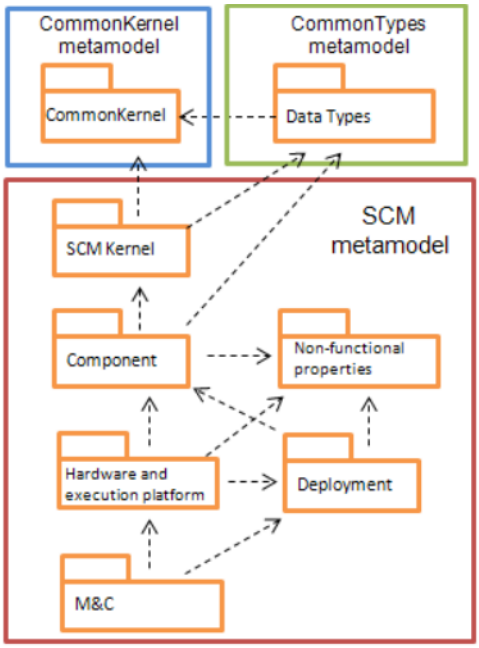


Figure 3.8.: Implementation of OSRA component model in the reference implementation

Source: [33]

`CommonKernel` Defines the basic entities that are used as the base elements of the language architecture.

`DataTypes` Defines all the possible data and the data types that can be used in an OBSW model.

`SCM Kernel` Defines the infrastructural part of the Space Component Model (SCM) which can be considered as the language to express all the concerns expressed by an OBSW model.

`Component` Defines a complete set of interfacing features (interfaces, events, datasets), component types implementations and interface ports.

`Non-functional properties` Defines the non-functional properties that can be applied to the modeling entities and defines a new language called the Value Specification Language (VSL) to specify values characterized by the measurement units.

`Deployment` Defines instantiation and deployment entities such as component instances, connection between them and their deployment on the hardware architecture.

`Monitor and Control (M & C)` Defines the means to specify the technical properties related to M & C that shall be provided in the OBSW model.

`Hardware execution platform` Defines entities related to the execution platform, Time Space Partitioning (TSP) and the hardware architecture.

3.5. OSRA SCM Model Editor

The toolset that the software architect can use to build OBSW models is organized as a set of Eclipse features and Eclipse plugins [10].

The toolset is available as [10]:

- A pre-installed Eclipse (Eclipse Neon) for Windows 64-bit
- An update site which consists of a set of static files which can be placed locally, on a web-server or on a file-server.

In the latter case, the software architect would have to use the Eclipse Update Manager to install the plugins [10].

Figure 3.9 shows a screenshot of the OSRA SCM model editor.

In line with the design flow and design views explained in section Section 3.3, different OSRA diagrams can be created with the help of OSRA SCM Model editor namely:

Interfaces, Events and Datasets diagram This is the first diagram of the OSRA activity and allows to define the data types, events, data sets and the interfaces that would be used by the components in the Component types diagram.

Component Types diagram This is the second diagram of the OSRA activity and it allows to define the component types, device types, execution platform service types, partition proxy types, required ports whose implementation would be used by the component instances diagram.

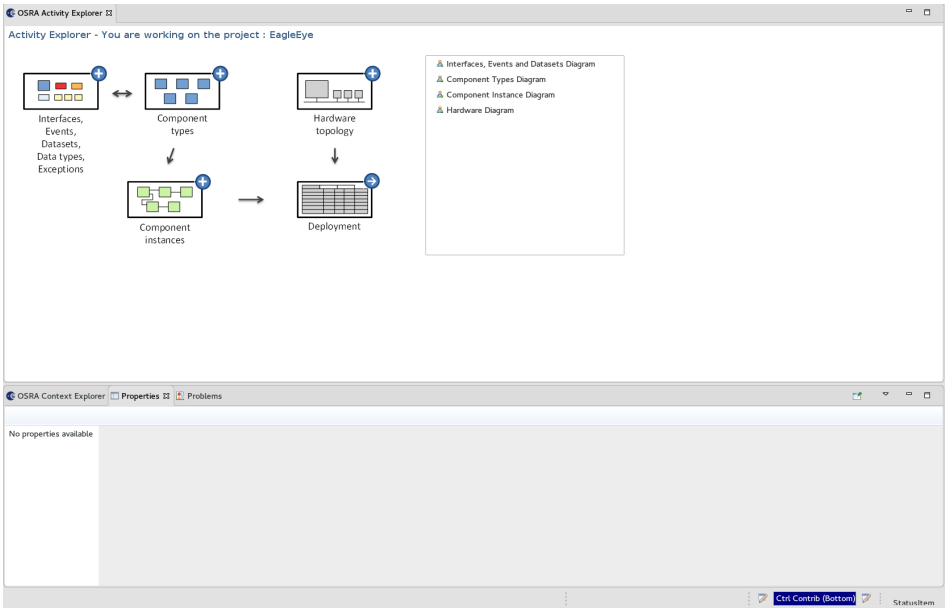


Figure 3.9.: Screenshot of the OSRA SCM model editor

Component Instances diagram This is the third diagram of the OSRA activity and it allows to define the component instances, device instances, execution platform service instances, partition proxy instances, provided interface slots, data receiver slots and event receiver slots.

Hardware diagram This is the last diagram of the OSRA activity and it allows to define the hardware elements such as processor boards, mass memory units, devices, buses.

There are also tables which are provided for diagram elements (if applicable) and they are usually found as tabs in pop-up window associated with the group that the element belongs to [10]. Tables are one of those classical tables where rows represents an element and each column represent a potentially computed property of the element. Rows can also contain sub-rows recursively which represent the sub-elements and the software architect can collapse or expand these sub-elements as desired. More information and details about install requirements and procedure, usage of the OSRA editor can be found in [10].

Chapter 4

Tasking Framework

4.1. Introduction

Future unmanned space missions have a great demand on computing resources for the on-board data processing or for control algorithms. Also, future space missions are requested to achieve more and more challenging scientific goals [31]. Besides the pre-processing of scientific data to reduce the data amount for the downlink, it is also necessary to handle the control systems with optical sensors, which come into play, for example in extra terrestrial navigation and landing systems [43]. Space missions like the Rosetta mission or the landing of the Mars rover Curiosity were based on pre-defined timed command lists to control the landing [22]. Because of this and the uncertainty of propulsion and parachute maneuver, the amount of different landing targets with low risks were considerably reduced [22]. But the interesting areas of planetary research might also include risky landing areas and hence an autonomous control is needed for the spacecraft to control the trajectory and there is also a need to integrate hazard avoiding algorithms [43]. However, these algorithms have a huge demand for computing power [22].

In TET-1 satellite mission (Technology demonstrator) and Bi-spectral Infrared Detection (BIRD) missions, the estimator and predictor modules were computed in a fixed order and fixed time in the control-cycle [23]. The timing was a combination of sensor latency and an additional gap time to satisfy the availability of data for computation and this led to a scant timing problem for the control torque computation due to over-estimated static safe-gap times [23][21]. During the Launch and Early Orbit Phase (LEOP), a timing violation in another bus application resulted in changing the the order of inter-dependent computations and corrupted data, which further resulted in an unexpected Attitude and Orbit Control System (AOCS) state [23]. Figure 4.1 A) depicts such a situation and it can be observed that `ea = E(A)` is calculated before `eb = E(B)`

Also, the current on-board systems for the space environment do not provide the needed computing power [22]. The space systems offer several controller boards on the spacecraft, most of them dedicated to only one subsystem and often twice for cold and hot redundancy. Such designs usually raise the power consumption and increase budgets like the mass, envelop and cost [22]. Hence a concept, which allows sharing of computing resources based on predefined configurations for different flight phases and fault scenarios is necessary [22]. The Tasking framework is an incarnation of the Inversion of Control design pattern which is popular practice in lightweight container frameworks [12].

A Tasking Framework is hence developed where the timing behavior is changed. Instead of starting computations at a predefined time in the computation cycle, a computation is started whenever the required information is available. All information are stored in messages distributed by channels and the channels initiate the computation when all the defined conditions for the computation are met [23][21]. The timing which can be achieved with Tasking Framework is as shown in Figure 4.1 B).

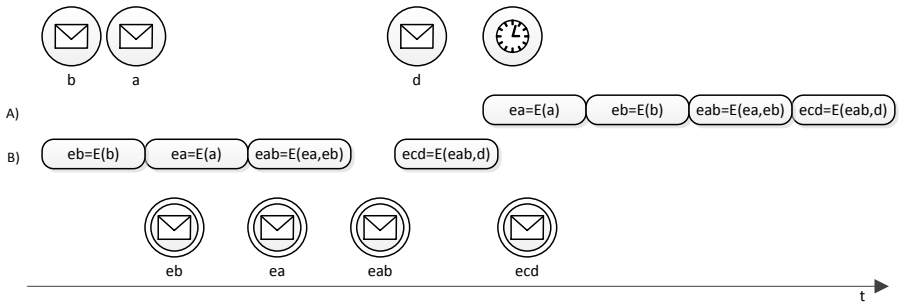


Figure 4.1.: Order and timing of computation tasks in BIRD and TET-1 vs order and timing of computation tasks in Eu:CROPIS

Source: [21]

4.2. Usage of Tasking Framework

The Tasking framework is based on C++ and provides some virtual base classes, which the application developer can overload to develop application specific computations. The Tasking Framework is designed to split the computations into small pieces, which are called tasks and they can be scheduled on the availability of the input data

For each task, a number of task inputs can be specified and each input can be associated with a task channel which provides the memory space and the synchronization for messages consumed by the task. In order to start a task, the task input needs to be configured with the expected number of data pushes on the associated task channel and when the number of pushes on the task channel meets the expectation set, the respective task input is activated. When all the inputs of a particular task are activated, the task is automatically started by the scheduler of the Tasking Framework, provided a free computing core is available. If no computing core is available, then the task is queued for execution.

Any of the task inputs of a particular task can be marked as final and when such an input is activated, the respective task is started immediately by the scheduler irrespective of the activation states of the other task inputs. As a result, the task can push onto another channel, which can trigger the other task associated with this channel. This leads to a kind of behavior similar to petri nets [38], where the activation of the task input is a token and the task execution is a transition [22]. The inputs associated to a task are reset when all its inputs are activated or when the respective activated input is marked as final. Such a reset operation on a task input sets the number of arrived data items on the associated task channel back to zero.

To specify timings in the Tasking framework, a special channel called 'event' is provided [22]. An event is associated with a clock and the task input to which it is associated is notified on each clock tick. When the required number of notifications match the expected number which is already set, the attached task input is activated. A Task event can be configured to work with absolute timing, i.e. the associated task input and in-turn the task itself is activated at fixed points in time, or a task event can be configured to work with a relative timing i.e, the task input and in-turn the task is activated at points in time relative to the execution time of the task.

To support mapping of tasks in a distributed system, task channels are associated with interfaces to read and write from and to networks and devices. These associations are set up by the configuration manager and are not visible to the application developer [22].

The current implementation of Tasking framework sits on top of Linux Portable Operating System Interface (POSIX) library, composed by a real-time clock interface, signaling mechanism, memory access and the tasks scheduler. The Tasking framework can also run on operating systems like Real-Time Executive for Multiprocessor Systems (RTEMS) and FreeRTOS using the outpost libraries which collectively provide a high level abstraction of the underlying operating system [22].

4.3. Use cases for Tasking Framework

In the project On-Board Computer - Next Generation (OBC-NG) by Deutsches Zentrum für Luft- und Raumfahrt (DLR), a decision was made to design the on-board computer systems as a combination of space qualified processing node, Commercial-off-the-shelf (COTS) processing nodes and network nodes [22]. As an operating system for this project, an enhancement of Realtime Onboard Dependable Operating System (RODOS) was used [22][37]. The enhancement covered mainly the support for multi-core and reconfigurable distributed systems [26] and the core element which made it possible was the Tasking Framework [22]. The configuration manager used in OBC-NG held predefined mappings of tasks and resources for different hardware configurations of computing resources and mission phases [37]. The communication infrastructure was set up based on the mappings during the configuration phases of the system.

The first usage of the Tasking framework was in the Autonomous Terrain-based Optical Navigation (ATON) project [43]. The project was about the navigation system for a moon landing scenario. The project showed that the Tasking framework was a useful way for the parallelization of computations in an expected manner [43].

Another use-case of Tasking Framework is in the AOCS of Euglena Combined Regenerative Organic food Production In Space (Eu:CROPIS) mission [21]. Eu:CROPIS uses a porting of the Tasking framework from Linux to outpost libraries which collectively act as an operating system Application Programming Interface (API) on top of RTEMS [21].

Tasking framework is also used in project named Matterwave Intrferometer in Microgravity (MAIUS) which deals with activities to demonstrate Bose-Einstein condensation and atom interferometry with rubidium and potassium atoms on a sounding rocket [21][15].

4.4. Use of Tasking Framework in this Master thesis

Tasking framework is used as a computational model, as discussed in the previous chapter. The reasons for adopting Tasking Framework in this Master thesis are the following:

- The Tasking framework guarantees that the timing behavior of the system is deterministic and amenable to static analysis.
- The Tasking framework has been proven to be expressive enough to handle real-world application as discussed in the previous section.

- Tasks do not interact with each other directly, but their communications are mediated by protected objects (task channels). These channels are shared resources equipped with a synchronization protocol in the form of priority based scheduling and First-In First-Out (FIFO) scheduling for tasks with same priorities which uses these shared resources [22].

However, during the course of this Master thesis, certain short-comings of Tasking framework have however been identified and can be found in Section 8.2 of Chapter 8.

Chapter 5

A programming model for OSRA

5.1. Introduction

In the previous chapters we have seen the model-driven software development approach that was centered on component-based techniques. Dijkstra's principle of separation of concerns was one of the cornerstone principles which was part of the software reference architecture and the proposed component model [33][34]. According to it, the user design space should be limited to the internals of the components, where only strictly sequential code can be used and the extra non-functional requirements are declaratively specified in the form of annotations on the component provided interfaces. This is already explained in detail in Step 7 (Specification of non-functional attributes) of Section 3.2 in Chapter 3.

As discussed in the previous chapters, the reference software architecture is made up of a component model, a computational model, a programming model and a conforming execution platform. It is also clear that the component model should be statically bound to a computational model to formally define the computational entities and the rules which govern their usage.

The realization of extra functional properties or more precisely, the generation of the complete infrastructure code can be done in two steps:

- Automated generation of the non-functional code i.e., the code for handling concurrency and interaction requirements for communication between components and the skeletons for the components themselves.
- Automated generation of containers for components and the connectors between components.

A code generator needs to be developed for this purpose and the next few chapters would be concerned about realizing the above mentioned steps. As a result, the third-party software supplier can then solely concentrate on implementing the functional code of the components. This is in line with the principle of separation of concerns, which is of very high interest.

The Automated proof-based System and Software Engineering for Real-Time systems (ASSERT) aimed at the definition of a MDE design process for the development of on-board software for satellites, cenetered on the priciples of correctness-by-construction and separation of concerns [31]. The cornerstone of the project was the adoption of an infrastrcuture called the Ravenscar Computational Model (RCM) [31]. This modeling infrastructure which was developed at the University of Padua [3] included a graphical modeling language and its editor, a model validator and a set of model transformations to feed the model-based schedulability analysis and code generation [3]. It is important to note that the ASSERT project lacked an explicit notion of a component model but incporporated a computational model, a programming model and a conforming execution platform [31]. Certain Ada Ravenscar Profile compliant code archetypes were developed to complete the formulation of the programming model and these archetypes adhered to the vision of principle of separation of concerns and amenable to code generation [35]. The Ada Ravenscar Profile basically does not allow any Ada language constructs that are exposed to unbounded execution-time and non-determinism [4]. The code archetypes used Ada run-time and hence could fit the needs of typical embedded systems which were resource-constrained [4].

In the following Artemis JU CHESS project, an initiative from ESA in parallel to the development of the SCM [33][31], a computational model named as Ravenscar Computational Model (RP in the following text) was chosen as the computational model [3]. RP directly emanated from the Ada Ravenscar Profile in language-neutral terms [35][34]. The code archetypes from the ASSERT project were revised and extended in the CHESS project and they as well targeted the Ada Ravenscar Profile, for the additional reason that the reduced tasking model used in the Ada Ravenscar Profile matched the semantic assumptions and communication model of real-time theory, the response-time analysis in particular [35]. The code archetypes developed in the CHESS project [34] are taken as reference for developing a programming model in this chapter. The code archetypes discussed in this Master thesis, however target the Tasking framework which is the chosen computational model for this Master thesis. The reasons for choosing Tasking framework as a computational model is already explained in Section 4.4 of the previous chapter. The code archetypes discussed in this chapter are first steps towards generation of the complete infrastructural code.

5.2. Structure of the code archetypes

The code archetypes discussed in this Master thesis take reference of the code archetypes in [34] and strive to attain as much separation as possible between the functional and extra-functional/non-functional concerns. At the implementation level, functional/algorithmic code of a component is separated from the code that manages the realization of the extra-functional requirements like tasking, synchronization and different time-related aspects.

A library of sequential code, which may have as many cohesive operations as the software supplier wishes to include in a single executing component, is included in a closed structure such as a component implementation. The mapping of this structure to the actual design entity of the infrastructural code is not of concern in this chapter and is handled in the next chapter. The sequential code in this structure is executed by a distinct flow of control of the system. The dedicated flow of control can be an active task, together with other tasking primitives from the Tasking framework (if the desired concurrency kind is `asynchronous`). Or, it can be a simple synchronous method/operation invocation, which uses the flow of control of the component requesting the service from outside (if the desired concurrency kind is `synchronous`). This leads to a combined effect that the component internals are completely hidden from outside environment, and the provided services invoked by the external clients are executed with the desired interaction semantics.

As multiple clients may independently require a range of services to be executed by one of the two desired flow of controls i.e., `synchronous` or `asynchronous`, it is necessary to safeguard these execution requests from mixing up. Safeguarding the execution requests in case of synchronous service requests is implicit as these requests would have been raised in the flow of control of the respective component asking for the service, but the safeguarding of execution requests in case of asynchronous requests needs to be explicitly handled and the way to achieve this is explained in the next parts of this section.

Service requests can often lead to valid/invalid results that need to be sent back safely to the components which made the requests. The service requester also need to be informed about any exceptions that might arise due to any unexpected situations during the servicing of the requests. The mechanisms and semantics necessary for realizing these requirements are also explained in the next parts of this section.

5.2.1. Synchronous release patterns

The archetypes for a synchronous release pattern are quite straight forward. When a request for a service is made with the desired concurrency kind specified as `synchronous`, the request is handled straight-away as a normal function/operation call in the flow of control of the service requester as shown in Figure 5.1. The results (if any) from the service requests, and exceptions (if any) during the course of handling the service requests are returned back to the service requester using the same flow of control as shown in Figure 5.1.

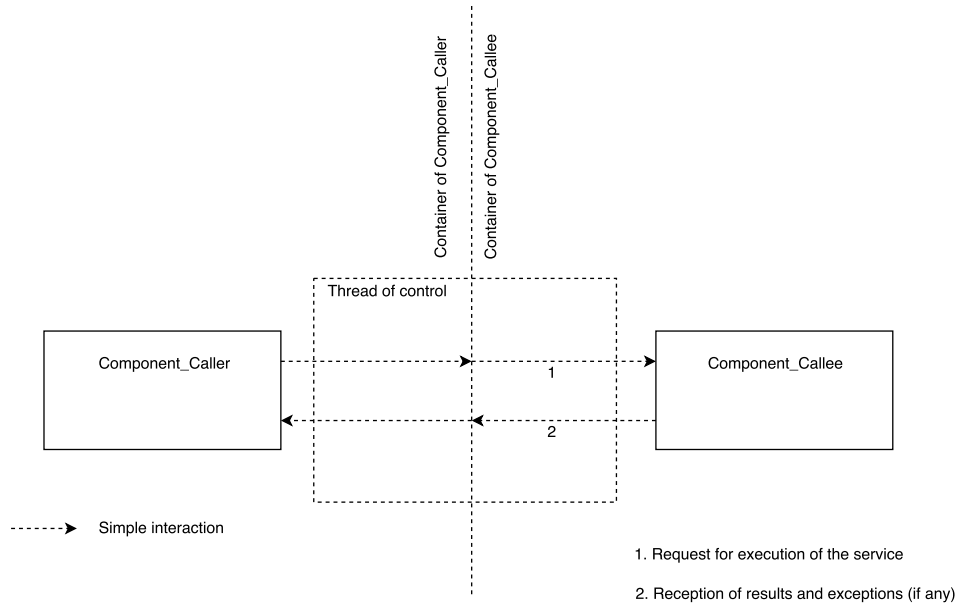


Figure 5.1.: Synchronous release pattern

5.2.1.1. **Protected**

When the non-functional property set on the service in the provided interface side of the component offering the service is `Protected`, it is necessary for the container that wraps around the component to safeguard this non-functional property. As the container is the entity that promotes the provided interface of the component, it intercepts the function/operation call from outside and provides exclusive access to the service implemented by the component. Semaphores provided by the Tasking framework is used for this purpose.

5.2.1.2. **Unprotected**

When the non-functional property set on the service in the provided interface side of the component offering the service is `Unprotected`, the semantics of handling the service request is essentially the same as the way the protected operations are handled, except for the fact that obtaining and releasing of the semaphore for the operation is not anymore needed.

5.2.2. Asynchronous release patterns

The archetypes for asynchronous release patterns are quite complicated when compared to archetypes for synchronous release patterns. As the requests cannot be anymore handled in the flow of control of the service requester in case of asynchronous release pattern, tasks from Tasking framework along with other tasking primitives, which are independent threads of execution can be activated to cater to these requests on the provided interface side.

The asynchronous service request is initially intercepted at the required interface port subsumed by the container of the component which makes the request. Here the data (if any), associated with the request is packaged and the packaged data is forwarded to the provided interface port, which is promoted by the container of the component handling the request. Along with the data associated with the service request, it is also important that the required interface port packages information (typically a function pointer as in C++) indicating how to send the results and exceptions (if any) back to the service requester.

In case of asynchronous service requests, if the service request leads to an exception being raised at the service provider end, it is the responsibility of the service requester to cope up with the exception and it is of no interest in this Master thesis. This Master thesis, however, explains how the information about the raised exception needs to be delivered to the service requester.

Each thread of control, having its own structure as explained above, is responsible for only one operation in the provided interface side of the component. As the release patterns for requests are already decided statically and as these release patterns are not expected to change at run-time, the number of threads of control that will be necessary to handle the service requests will be known at compile-time.

This is very similar to the way asynchronous release patterns are handled in the code archetypes listed in [35][34] except for the fact that they do not consider service

requests which might result in results or exceptions that need to be sent back to the service requester [35].

5.2.2.1. **Sporadic**

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Sporadic`, it is the responsibility of the container of the component providing the service to safeguard this property. The sporadic property requires that two subsequent requests for the service needs to always be separated by no less but possibly more than a minimum guaranteed time span, known as the MIAT (Minimum Inter-Arrival Time) [32][33]. The container makes use of tasking primitives such as a task channel, task event and a task from the Tasking framework for this purpose.

The general structure of the thread of control on the service provider end, necessary to handle sporadic service requests consists of a task with two synchronized task inputs, attached to the task as shown in Figure 5.2. The task inputs are not marked as final. One of the task inputs is associated with a task event, with absolute timing (fixed task wake-up times) and the other task input is associated with a normal task channel. The task event is configured to wake up the task periodically after every MIAT interval. The task input associated with a normal task channel is configured so that the task input is activated as soon as a push is made against its associated task channel. This task then is instantiated in the container of the service provider component.

When a provided interface port, promoted by the container of the component handling the request, receives a sporadic service release request, it intercepts the request and pushes the packaged data against the channel associated with the task.

Because the task inputs are not marked as final, the task is activated only after both its task inputs are activated. When activated, the functions of the task will then be to:

Step 1 Unpack the packaged data

Step 2 Acquire the semaphore provided by the Tasking framework associated with the service

Step 3 Execute the desired service

Step 4 Reset the task event attached to the task

Step 5 Release the semaphore acquired

Step 6 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port.

In this way, the non-functional properties associated with an asynchronous sporadic release pattern can be preserved at run-time.

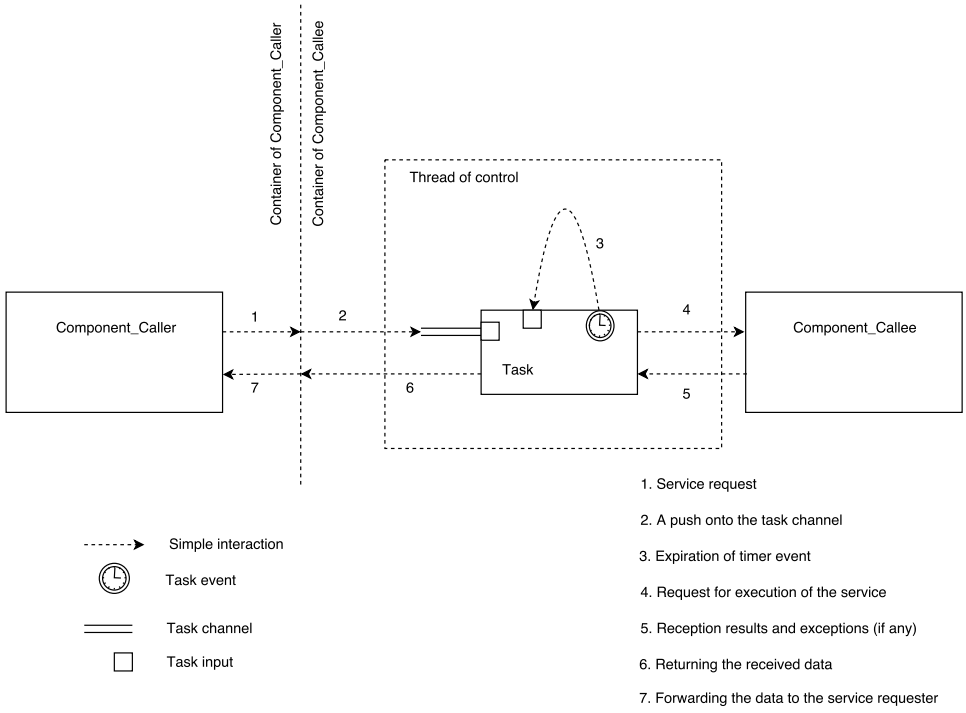


Figure 5.2.: Thread of control for asynchronous sporadic release pattern

5.2.2.2. **Protected**

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Protected`, it is the responsibility of the container, of the component providing the service, to safeguard this property. The container makes use of tasking primitives such as a task channel and a task from the Tasking framework for this purpose.

The general structure of the thread of control on the service provider end, necessary to handle this kind of service request, is a task with one task input attached to the task as shown in Figure 5.3. The task input is configured so that the task input is activated as soon as a push is made against its associated task channel.

When a provided interface port, promoted by the container of the component handling the request, receives a service release request of this kind, it intercepts the request and pushes the packaged data against the channel associated with the task. The task is then activated and the functions of the task will then be to:

Step 1 Unpack the packaged data.

Step 2 Acquire the semaphore provided by the Tasking framework associated with the service.

Step 3 Execute the desired service.

Step 4 Release the semaphore acquired.

Step 5 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port.

In this way, the non-functional properties associated with an asynchronous protected release pattern can be preserved at run-time.

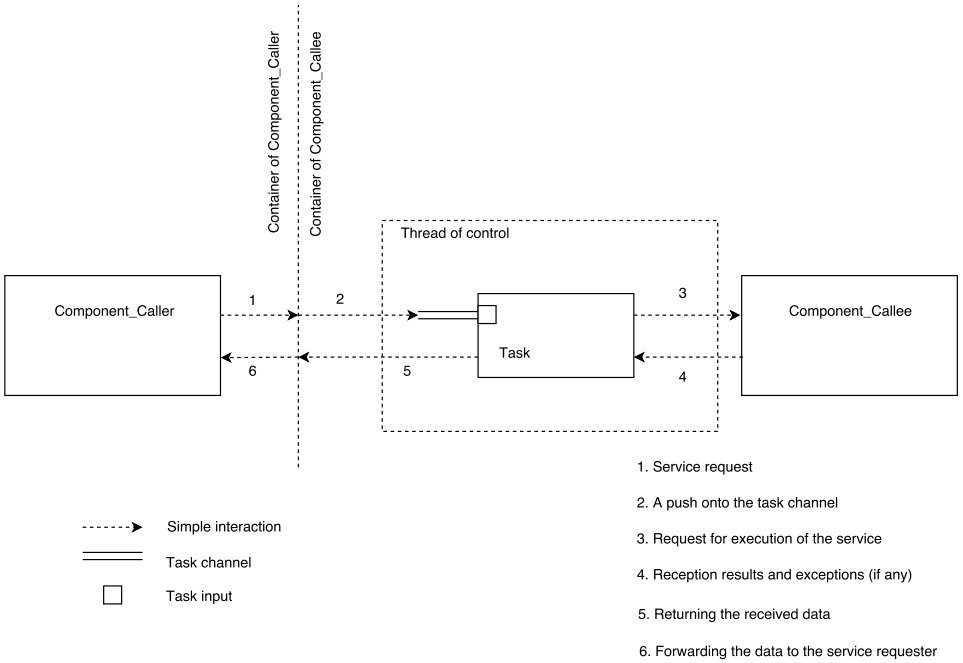


Figure 5.3.: Thread of control for asynchronous protected release pattern

5.2.2.3. **Bursty**

When the non-functional property set for the handling of the service on the provided interface side of the component offering the service is `Bursty`, it is the responsibility of the container, of the component providing the service, to safeguard this property. The bursty property requires that a service can be activated at most a given number of times in a given interval called the Bound Interval (BI) [32][33].

The general structure of the thread of control on the service provider end, necessary to handle this kind of service request, is a task with two non-synchronized task inputs attached to it as shown in Figure 5.4. The task inputs are marked as final. One of the task inputs is associated with a task event, with absolute timing (fixed task wake-up times) and the other task input is associated with a normal task channel. The task event is configured to wake up the task periodically after every bound interval. The task input associated with a normal task channel is marked as `final` indicating that the task input is activated as soon as a push is made against its associated task channel. The task also has an internal counting semaphore provided by the Tasking framework in order to keep a count of the number of service requests handled within the bound interval.

When a provided interface port, promoted by the container of the component handling the request, receives a service release request with bursty nature, it intercepts the request and pushes the packaged data against the channel associated with the task.

Because the task inputs are marked as final, the task is activated if any one of its task inputs are activated. When activated, the functions of the task will then be to:

Step 1 Check the activated input. If the activated input is the one that is attached to a task event, then replenish the counting semaphore, restart the attached task event and go to step 8.

Step 2 Unpack the packaged data.

Step 3 Acquire the counting semaphore local to the task which is used to enforce the max. number of activations within a bound interval.

Step 4 Acquire the semaphore provided by the Tasking framework associated with the service.

Step 5 Execute the desired service.

Step 6 Release the semaphore associated with the service.

Step 7 Return the results and the exceptions associated with the service request back to the service requester making use of the information of the service requester packaged by the required interface port.

In this way, the non-functional properties associated with an asynchronous bursty release pattern can be preserved at run-time. It is important to note that the code archetypes which were developed for the CHESS project do not mention the scheme to handle this kind of release pattern [35][34].

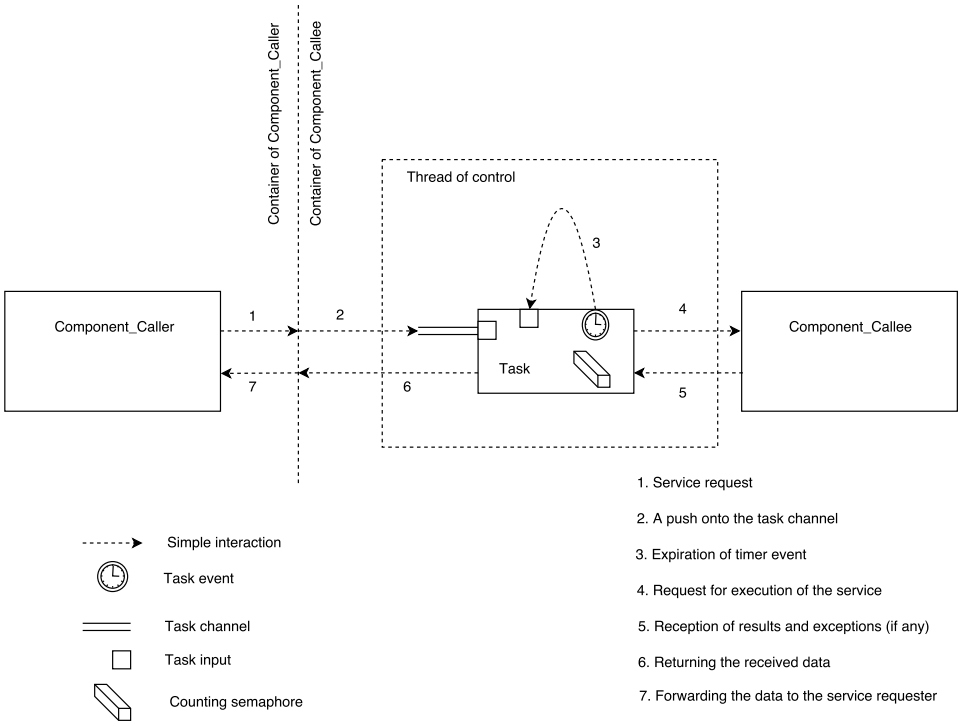


Figure 5.4.: Thread of control for asynchronous bursty release pattern

5.2.2.4. **Cyclic**

When the non-functional property set for a service on the provided interface side of the component is `Cyclic`, it is the responsibility of the container of the component, to safeguard this property. Cyclic property requires that the associated service be activated periodically and with a non-zero initial offset [32][33].

The general structure of the thread of control, necessary to handle this kind of non-functional property is a task with a task event, provided by the Tasking framework, attached to it as shown in Figure 5.5. The task event is configured to wake up the associated task periodically, with absolute timing (fixed task wake-up times). The task event can also be configured to wake up the associated task for the very first time with an initial offset.

When a provided interface port, promoted by the container of the component has a service which needs to be activated periodically, the task is activated and it performs the following functions:

Step 1 Acquire the semaphore provided by the Tasking framework associated with the service.

Step 2 Execute the desired service.

Step 3 Release the semaphore acquired.

The service with a cyclic nature cannot be requested from an external component [32]. The services also need to be parameterless and cannot send out results or throw exceptions [32].

In this way, the non-functional properties associated with an asynchronous cyclic release pattern can be preserved at run-time.

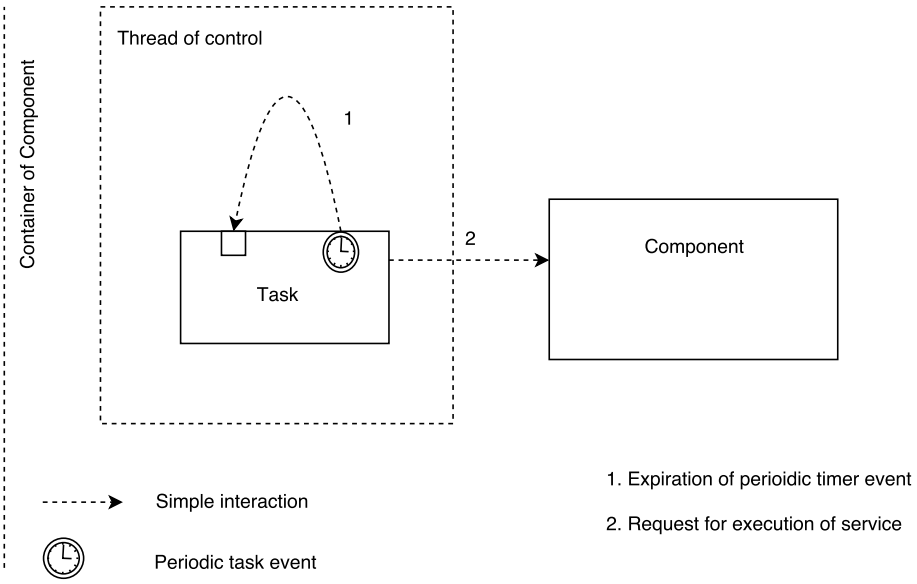


Figure 5.5.: Thread of control for asynchronous cyclic release pattern

Chapter 6

Infrastructural code generation

6.1. Introduction

After designing an OBSW model using the OSRA SCM model editor and following the component-based software development approach that comes with it, the OBSW model entities need to be mapped to infrastructure code. The reference programming model for OSRA, discussed in the previous chapter helps us in progressing towards this goal. But, it is necessary to understand the overall design approach for the generated code and present the abstractions that will be offered to the software supplier. This chapter, deals with these things in detail. Similar efforts from the Artemis JU CHESS project [34], provide the perfect base for discussions in this chapter of the Master thesis.

6.2. User model entities in the Platform Independent Model (PIM) phase

A detailed description of all the modeling entities that the software architect can use, can be found in the specification of the metamodel for the OSRA component model [32]. However a brief description of them is noteworthy here:

Datatypes The software architect can create a set of project-specific data types and constants using the `Data Types` language unit of the `CommonTypes` metamodel and the language unit is designed to provide the software architect an expressive power comparable to the languages with strong types (e.g. Ada) [32]. The supported type definitions are boolean types, integer types, float types, enumeration types, fixed point types, array types, structured types, string types, union types, alias types, opaque types, external types and unconstrained types. Some of these data type

definitions are obvious for readers with programming skills in typed languages such as Ada, C or C++.

Interface An interface is a specification of a coherent set of services and it represents the definition of a contract. An interface is defined independently of the entities implementing it (e.g. Component type). An interface may enlist declaration of operations, which are the functional services that shall be offered by the entities implementing it. The services include a name, a set of ordered parameters and one or more exceptions that they might throw when things go wrong during the handling of the service. Parameters are typed with one of the types mentioned above and have a mode (`in, out` or `inout`). A component type may expose one or more interfaces and the same interface can be exposed by different component types. An interface may also contain the declaration of one or more interface attributes, which are the parameters that are accessible via the interface implementations.

Component type A component type is an entity which specifies the external interfaces of a software component which are defined in isolation and are used to declare relationships with the other components and system in general. It conforms to the principle of encapsulation and as a consequence, all the interactions with other components are performed exclusively via its explicitly declared interface. A component type usually encompasses:

- A definite number of provided interface ports
- A definite number of required interface ports
- A definite number of dataset emitter ports
- A definite number of dataset receiver ports
- A definite number of event emitter ports
- A definite number of event receiver ports

Component implementation It is an entity that represents a concrete realization of a component type. It is functionally identical to the component type, except that the source code is added to the component implementation and may also define number of component implementation attributes.

Component instance It is an instantiation of a component implementation and hence contains all the instantiations of the structural features, such as provided and required interface ports. It also contains instantiation of all attributes (interface attributes, component type attributes and component implementation attributes). It is also the elementary deployment unit for the OBSW model [32].

6.3. Mapping of design entities to the infrastructural code

As the generated code should target the Tasking framework, which is the target computational model in this Master thesis and because the Tasking framework is written in C++, the following sections explains the mapping of design entities to the infrastructure code that will be generated in C++.

On analyzing the specifications of the metamodel for the OSRA component model [32], it is clear that there are different corner cases that can arise during the construction of the OBSW models using OSRA component model and it is necessary that these corner cases are effectively handled in the software design for the infrastructural code. The following sections try to build an OBSW model keeping the the corners cases in mind and attempt to explain the overall design approach.

6.3.1. Corner cases arising during the construction of OBSW model using OSRA component model

The different corner cases which can arise are:

- Multiple provided interfaces which refer to the same interface type are promoted by the container of a component
- Multiple required interfaces which refer to the same interface type are subsumed by the container of a component
- Multiple interfaces provide exact same operations

The first and second corner cases are handled in the following example. But, the other case will be treated directly in the later section, which deals with the software design for the generated infrastructure code.

6.3.2. An example OBSW model

Our simple OBSW model, yet effective to serve the intended purpose, is built as per the proposed component-based development approach explained in the section Section 3.2 in Chapter 3. As already mentioned in that section, the component-based approach puts a lot of emphasis on the definition of component interfaces [33] and it is followed here as well. Components are built from scratch using newly defined interfaces. All model entities defined here are instantiations of the modeling entities specified in the metamodel [32]. The OBSW model is designed using the OSRA SCM model editor

mentioned in Section 3.5 in chapter Chapter 3. The model entities from the OSRA SCM model editor can be exported as images and they are used in this sub-section for illustration purposes.

In this simple example, two simple components namely `Caller` and `Callee` are designed. The first component requests for a service `OperationAdd` which can add two numbers, as the name suggests and this service is implemented in the second component. Another service named `CallOperationAdd` is implemented in the first component, which can be requested to trigger the OBSW model execution on a periodic basis. Different non-functional properties, as explained later in this section, are strewn on the required and provided interfaces of these components to make the example a bit more interesting and also capture the above mentioned first and second corner cases in the example.

Step 1: Definition of data types and events As the Master thesis requires to emphasize more on effectively capturing interactions and concurrency semantics required for communication between the designed components, the data types chosen in this example are fairly simple. But it is important to note that the scheme of mapping of these simple data types to the infrastructural code (explained in the later sections), can be scaled to fairly complex data types as well. The data types, exception types and the event type used in this example are as shown in Figure 6.1.

- Two data types namely `FixedLengthStringType` and `IntegerType` of type `UNSIGNED` are defined and they are named `StringType` and `IntegerType` respectively.
- Three exception types, named `OperandException`, `MemoryException` and `OverflowException` are defined.
- An `Event` type, which can be used for asynchronous notifications [32] is instantiated and it is named as `FailureEvent`. Two event parameters are also instantiated as shown in Figure 6.1.

Step 2: Definition of interfaces Two interface namely `InterfaceA` and `InterfaceB` are designed as shown in Figure 6.1. `InterfaceA` has only one single operation called `CallOperationAdd` and `InterfaceB` has an operation called `OperationAdd` and an interface attribute of data type `IntegerType` called `m_StatusValue`.

- The operation `CallOperationAdd` is a parameterless operation and it is intended to be the service which can in turn request the service `OperationAd` which can add two numbers.
- The operation `OperationAdd` in `InterfaceB`, as the name suggests, is intended to be the service which can add two numbers, send back the results and raise

a pre-defined exception if necessary. It has three operation parameters and can throw different exceptions as mentioned in Figure 6.1.

- The interface attribute `StatusValue` in `InterfaceB` is of type `CFG` and it indicates that the interface attribute is a configurable parameter [32]. As a result, two operations for the purpose of setting and getting the values of the interface attribute need to be generated.

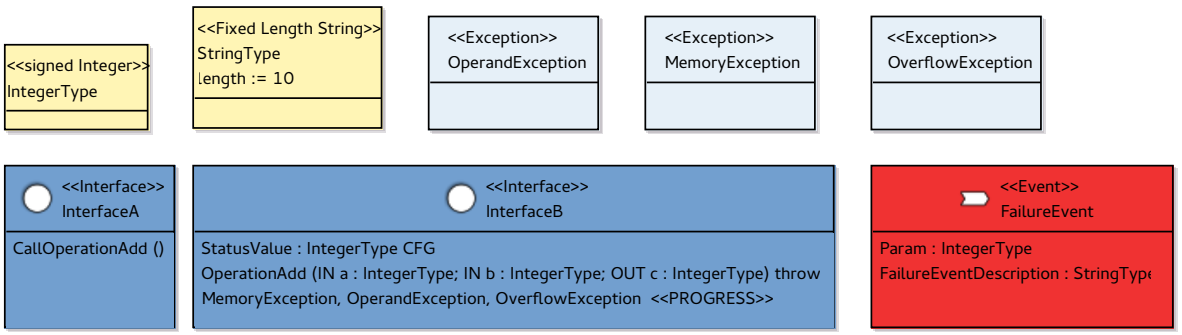


Figure 6.1.: Data types, events, exceptions and interfaces diagram

Step 3: Definition of component types Component types namely `Component_Caller` and `Component_Callee` which form the basis for a reusable software asset are defined as shown in Figure 6.2. In Figure 6.2, a circle without a fill on a component type denotes a provided interface port and a black filled-in circle on a component type denotes a required interface port. Similarly, an arrow without a fill on the component type denotes an event receiver port and a filled-in arrow on the component type denotes an event emitter port.

`Component_Caller` has:

- Provided interface port `ProvidedInterfacePort` which refers to `InterfaceA`.
- Required interface port `RequiredInterfacePortType1` which refers to `InterfaceB`.
- Required interface port `RequiredInterfacePortType2` which refers to `InterfaceB`.
- Event receiver port `FailureEventReceiverPort` which refers to `Failure Event`.

The desired interaction kind for the operations in the required interface ports of `Component_Caller` are as shown in Table 6.1

Table 6.1.: Desired interaction kind for operations in the required interface ports

Required interface ports	Operations	Interaction kind
RequiredInterfacePortType1	OperationAdd	synchronous
	Interface attribute setter	synchronous
	Interface attribute getter	synchronous
RequiredInterfacePortType2	OperationAdd	asynchronous
	Interface attribute setter	asynchronous
	Interface attribute getter	asynchronous

Component_Callee has:

- Provided interface port ProvidedInterfacePort1 which refers to InterfaceB.
- Provided interface port ProvidedInterfacePort2 which refers to InterfaceB.
- Event emitter port FailureEventEmitterPort which refers to FailureEvent.

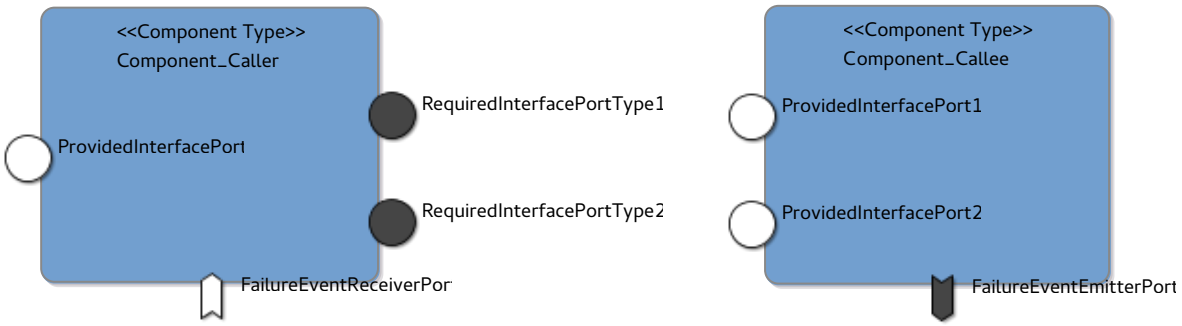


Figure 6.2.: Component types diagram

Step 4: Definition of component implementations Component implementations are created from the component types.

Component_Caller has one component implementation named as Component_Caller_impl and Component_Callee has one component implementation named as Component_Callee_impl. The component implementation Component_Callee_impl implements the means to store the attribute Param of InterfaceB, that is exposed through its provided interface ports, namely ProvidedInterfacePort1 and ProvidedInterfacePort2.

No maximum memory footprint for component implementations are defined or no detailed design activity of the component implementations are performed as they are not of concern in this Master thesis.

Table 6.2.: Non-functional property for the operation in the provided interface slot

Provided interface slot	Operation	Non-functional property
`ProvidedInterfaceSlot`	`CallOperationAdd`	`Cyclic`, Period = 2s

Step 5: Definition of component instances The component instances are the instances of component implementations [33].

Two component instances as shown in Figure 6.3 are defined, namely:

- `Component_Caller_inst` which is an instantiation of `Component_Caller_impl`.
- `Component_Callee_inst` which is an instantiation of `Component_Callee_impl`.

Step 6: Definition of component bindings Component bindings as shown in Figure 6.3 are defined:

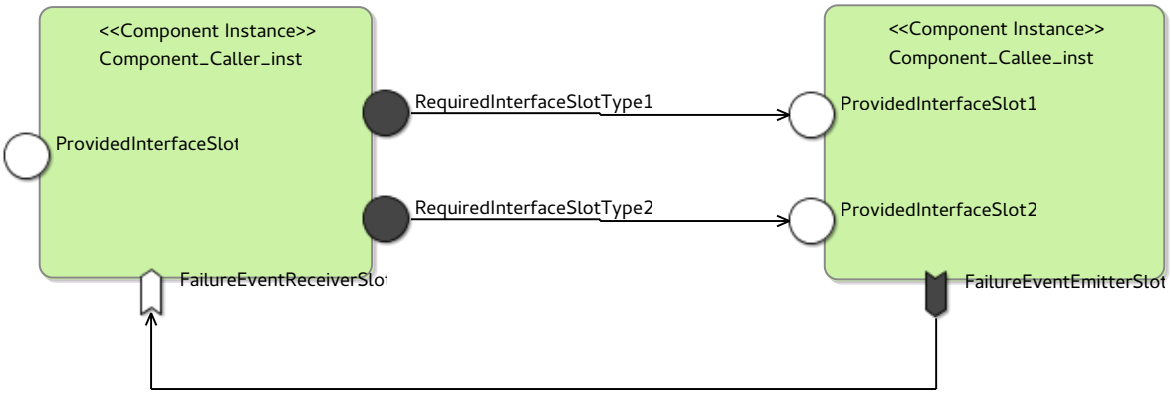


Figure 6.3.: Component instances diagram

Step 7: Specification of non-functional attributes The non-functional properties are defined on the component instances and the component bindings defined in the previous step. The non-functional properties language unit of the specification of a metamodel provides a Value Specification Language (VSL) unit, which permits the specification of the the non-functional properties qualified with a measurement unit [32]. The VSL is used here to define values of non-functional properties with a measurement unit.

For the provided interface slot in the component instance `Component_caller_inst`, a non-functional property is specified as shown in Table 6.2

For the provided interface slots in the component instance `Component_callee_inst`, non-functional properties are specified as shown in Table 6.3

Table 6.3.: Non-functional properties for the operations in the provided interface slots

Provided interface slots	Operations	Non-functional properties
ProvidedInterfaceSlot1	OperationAdd	Protected
	Interface attribute setter	Protected
	Interface attribute getter	Unprotected
ProvidedInterfaceSlot2	OperationAdd	Sporadic, MIAT = 2s
	Interface attribute setter	Protected
	Interface attribute getter	Protected

Table 6.4.: Non-functional property for event reception

Event receiver slot	Event	Non-functional property
FailureEventReceiverSlot	FailureEvent	Protected

It is important to note that the WCET and deadline values for the operations in the provided interface slots are not handled, as the safeguarding of these properties are not of concern in this Master thesis.

For the event receiver slot in the component instance Component_caller_inst, a non-functional property is specified as shown in Table 6.4.

Step 8: Definition of the physical architecture The hardware topology provides a description of the system hardware. As hardware modeling is not of concern in this Master thesis, a simple hardware topology as shown in Figure 6.4 is considered.

A processor board with a processor and a processor core is instantiated. Two connection docks are attached to the processor board and a bus is used to connect the connection docks. The component instances are deployed on the processor core and the component bindings are deployed on the bus.

This OBSW model is subjected to model validation against the OSRA Specification Compliance and the SCM meta-model compliance, in the OSRA SCM editor [10]. Only after the OBSW model is successfully validated, can the OBSW model be considered as a suitable candidate for automatic generation of infrastructure code [10].

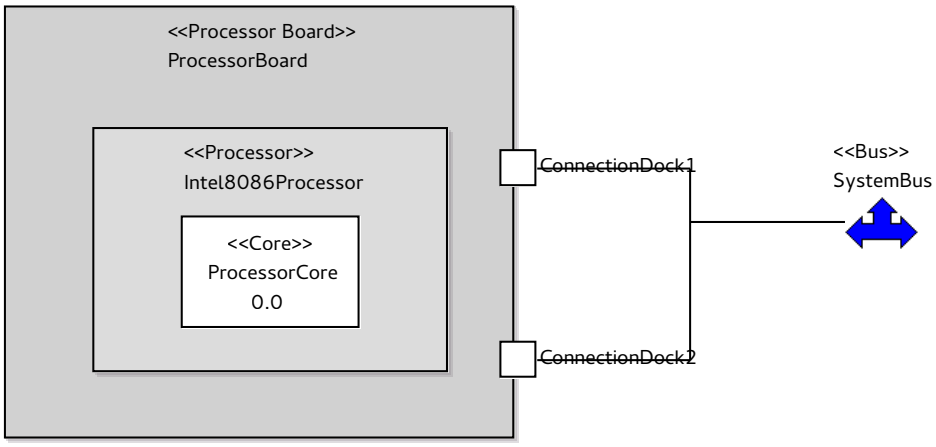


Figure 6.4.: Hardware topology diagram

6.3.3. Software design approach for the generated code

This section deals with the software design approach for the generated infrastructure code. Some of the necessary good characteristics of a software such as reusability, separation of concerns and minimal complexity are targeted already at the reference architecture level and it needs to be preserved at the generated software level as well. The other main characteristics of the generated software which are of utmost value are [24]:

Testability The software must be testable and there must be suitable constructs in the generated software which assist in writing automated unit tests or user driven tests to test it. In this Master thesis, it is taken care of that the generated C++ classes have corresponding abstract base classes. These abstract base classes are used in constructing mock classes which can be used for the purpose of testing each class independently [46]. More about this is explained in the next chapter which focuses on the results and further scope of this Master thesis.

Extensability and Refactorability The software generated must be loosely coupled so that the entire code base is more resilient to changes and extensions. Dependency injection software design pattern [12] is used wherever appropriate to make the generated software loosely coupled.

Portability The generated software must be portable across systems and environments. The data types used in the generated software should be portable across multiple platforms and the data type standardizations from C++11 are made use of in the generated software.

High fan-in The generated software is designed in a way that a particular C++ class is used by large number of other C++ classes.

Low-to-medium fan-out The generated software is designed in a way that a particular C++ class uses not many classes, so that the complexity of the generated software is not too high. Unfortunately, this characteristic is not completely respected in this software design, because the complexity of a particular class depends on the complexity of the corresponding model entity in the user model.

Unified Modeling Langauge (UML) class diagrams, wherever appropriate, are judiciously used in this section to show a high level representation of the generated C++ classes and datastructures.

Each of the following sub-sections, is divided into two parts:

- The first part throws light on the idea of how a mapping of a given model entity to an infrastructural code entity can be done.
- The second part makes the approach clear by taking the reference of our OBSW example model discussed in the previous section.

The standard notations from UML class diagrams are used. The following additional legends are introduced:

- A UML class in dotted notation means that the explanation for that particular has already been given in the previous sections or would be given later in the following sections
- A package notation from UML is used to indicate the namespaces that the respective classes in the UML diagram can be found

6.3.3.1. **Namespaces**

Namespaces from C++ (similar to packages in Ada as shown in [34] for the Artemis JU CHESS project) are used to differentiate component types, component implementations, etc. of different components. The names for the namespaces are obtained from the names of the component types in the OBSW model. The name of each component type is a namespace and this namespace holds all the code entities which are related to its corresponding component type. An additional namespace called `General` is used to define all the code entities which would be shared across all components.

For our example OBSW model: Three namespaces, namely `General`, `Component_Caller` and `Component_Callee` are created as shown in Figure 6.5

Figure 6.5.: UML class diagram representation for different namespaces in the example OBSW model

6.3.3.2. **Data types and events**

A data type from the OBSW model is translated into a simple `typedef` statement from C++ as shown in [41][28].

For our example OBSW model:

- The data type `IntegerType`, is translated to `typedef int8_t IntegerType`
- The data type `StringType`, is translated to `typedef std::string StringType`

A subset of all possible data types from the OSRA Component Model can be translated to simple `typedef` statements as shown above. More information about the subset of data types for which this successfully works is given in the next chapter.

The exception types from the OBSW models are translated into simple enumeration literals from C++ as shown in [11]. These exceptions, which can be thrown by a particular operation are grouped under an enumeration. This enumeration is further instantiated in a C++ struct as shown in [11].

For our example OBSW model: The three exceptions `OperandException`, `Memory Exception` and `OverflowException` are translated to enumeration literals. These exceptions can be thrown by `OperationAdd`, which is defined in `InterfaceB`. Hence the enumeration literals, corresponding to the exceptions, are stored together as an enumeration named `OperationAddException_InterfaceB` as shown in the Figure 6.6. This exception is further instantiated in a C++ struct `OperationAddReport_InterfaceB` as shown in the Figure 6.6

An event is similar to a message which is passed asynchronously from emitter to receiver [32] in an OBSW model. Because it resembles a message, it can be mapped to an abstract base class and a corresponding concrete implementation class as in [34]. As already explained, the abstract base classes help in improving the testability of the generated software. Appropriate setters and getters for the event parameters are declared as pure virtual methods in the abstract base class for the event and they are implemented in their corresponding concrete implementation.

For our example OBSW model: The `FailureEvent` is mapped as an abstract base class named `FailureEventInterface` and a concrete implementation class named

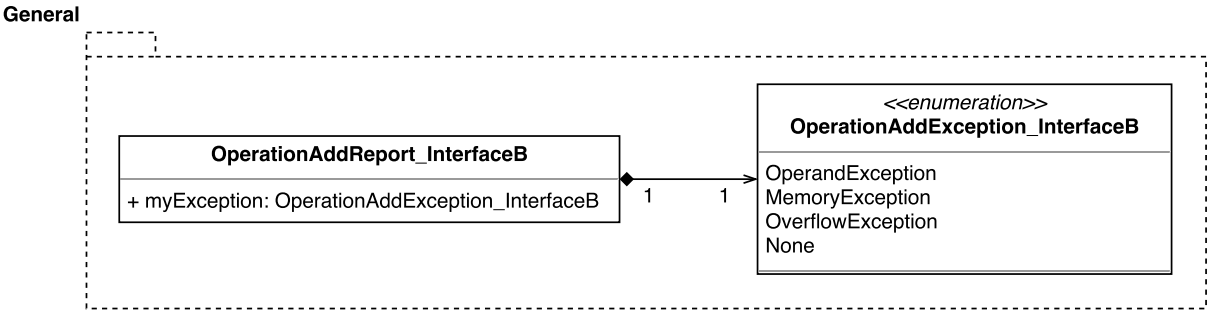


Figure 6.6.: UML class diagram representation for exceptions in the example OBSW model

`FailureEvent`. Appropriate setters and getters for the event parameters `Param` and `FailureEventDescription` are declared as pure virtual methods in the `FailureEvent Interface` abstract base class and implemented in the `FailureEvent` concrete implementation class. An UML class diagram representation of the generated classes is shown in Figure 6.7.

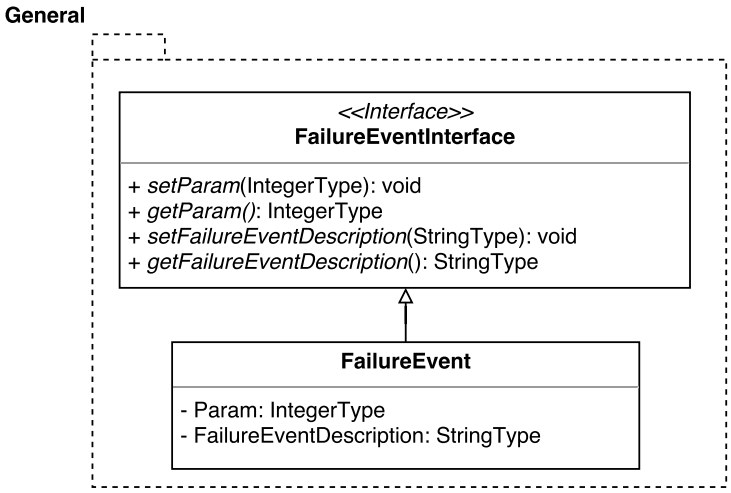


Figure 6.7.: UML class diagram representation for event in the example OBSW model

All the infrastructure code entities mentioned above are present in the namespace `General`.

6.3.3.3. **Interfaces**

An interface can be mapped to a C++ abstract base class, as in [34]. This abstract base class consists of the following entities:

- For each interface operation, a pure virtual method is declared in the interface class. The names and data types of the input parameters for this pure virtual method correspond to the names and data types of the interface operation parameters. The operation parameters, with `ParameterDirectionKind` as `in` are translated to constant references and the operation parameters with `ParameterDirectionKind` as `out` or `inout` are translated to plain references.
- For each interface attribute parameter of type `CFG`:
  - A public non-static class variable of name and data type corresponding to the name and data type of interface attribute is added.
  - Pure virtual setter and getter methods for the interface attributes are declared. The data types and names of the input parameters in the setter and getter methods mimic the name and data type of the interface attribute.
- For each interface attribute parameter of type `MIS`, which is fixed and is not variable [32]:
  - A `const` protected non-static class variable of name and data type corresponding to the name and data type of interface attribute is added.
  - No getter and setter methods are added.
- For each interface attribute of type `DAT`, which are modifiable by the component only and not by external entities [32]:
  - A protected non-static class variable of name and data type corresponding to the name and data type of interface attribute is added.
  - No getter and setter methods are added.

For our example OBSW model The C++ classes shown in Figure 6.8 are generated:

- `InterfaceA` along with the operation `CallOperationAdd` is mapped to an abstract base class `InterfaceA` with a pure virtual method `CallOperationAdd`.
- `InterfaceB` has one operation `OperationAdd` and one interface attribute parameter `StatusValue` of type `CFG`. These are mapped to an abstract base class named `InterfaceB` with the following pure virtual methods:
  - `OperationAdd` with two input parameters of type `const IntegerType&` and one input parameter of type `IntegerType&`.
  - getter method for the interface attribute `StatusValue` with an input parameter of type `IntegerType&`.

– setter method for the interface attribute `StatusValue` with an input parameter of type `const IntegerType&`.

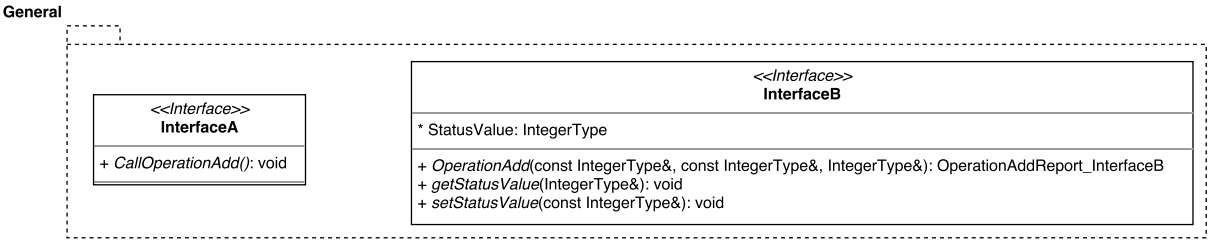


Figure 6.8.: UML class diagram representation for interfaces in the example OBSW model

Because of the corner case that multiple interfaces can have exactly same operations, it is necessary to refine these interfaces using the interface helper abstract base classes as shown in [17] and in [42]. UML class diagram as shown in Figure 6.9 explains such an approach in this context. In each interface helper class:

- Implementations for all the inherited pure virtual methods from the parent interface are provided.
- The implementations consist of simple method calls to new pure virtual methods.
- These new pure virtual methods have method signatures same as the pure virtual methods that are inherited and implemented. However, it is important to note that the names of these new pure virtual methods are different from the inherited pure virtual methods, as shown in the Listing 6.1.

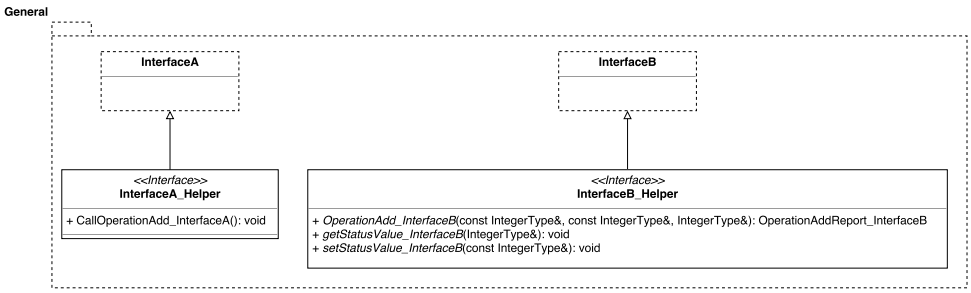


Figure 6.9.: UML class diagram representation for interface helpers in the example OBSW model

For our example OBSW model:

- `InterfaceA_Helper` is defined, which inherits from the interface `InterfaceA` and which implements the pure virtual method in the parent interface `InterfaceA`. The implementation contains a simple call to a new pure virtual method added to the original method name from the parent interface as shown in the Listing 6.1.

Listing 6.1 Code excerpt from the generated code for `InterfaceA_Helper`

```
class InterfaceA {
public:
      InterfaceA(){}
      virtual ~InterfaceA(){}
      virtual void CallOperationAdd(void) = 0;
};

class InterfaceA_Helper: public InterfaceA {
public:
      InterfaceA_Helper(){}
      virtual ~InterfaceA_Helper(){}
      virtual void CallOperationAdd_InterfaceA(void) = 0; //New pure virtual method
      virtual void CallOperationAdd(void)final {return (CallOperationAdd_InterfaceA());}
};
```

- `InterfaceB_Helper` is defined, which inherits from the interface `InterfaceB` and which implements all the pure virtual methods in the parent interface `InterfaceB`. Each implementation contain a simple call to the new pure virtual methods added. The `InterfaceB_Helper` class is designed and implemented the same way as `InterfaceA_Helper` is designed in the code excerpt in Listing 6.1

.

The combined effect is that now, more than one original parent interfaces (resembling model entities) can have same operations and interface attributes. The refined interfaces redefine the methods from the original parent interfaces, so that there are no confusions between operations from different interfaces. Of course, a straight forward solution would have been to incorporate the concept of namespaces from C++, but it is not suitable for this design and the reason is explained later in this section while discussing about mapping for component types.

For each interface operation and interface attribute in an interface, a C++ struct is defined to carry around the values of the operation parameters or the values of the interface attributes. These data structures come in handy, when the interface operations or interface attribute access operations need to be accessed asynchronously. The data structures also hold general purpose polymorphic function wrappers from C++11 standard as shown in [5] to store the call-back functions wherever appropriate.

For our example OBSW model:

- A struct `OperationAddStruct_InterfaceB` is defined as shown in Figure 6.10
- A struct `StatusValueStruct_InterfaceB` is defined as shown in Figure 6.10

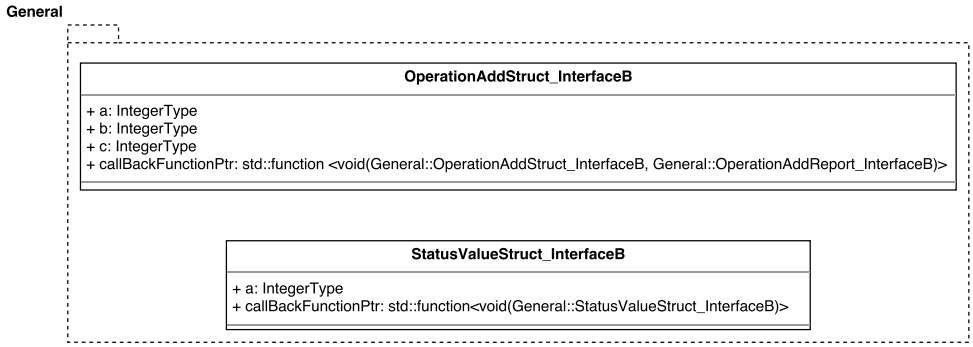


Figure 6.10.: UML class diagram representation of data structures for interface operation and interface attribute in the example OBSW model

All the infrastructure code entities mentioned above are present in the namespace `General`. They are placed in the namespace `General` because these structs are used by both `Caller` and `Callee` components.

6.3.3.4. **Parameter channels and parameter queues**

The data structures which are defined to carry around the values of the operation parameters or values of the interface attributes, need to be pushed onto parameter channels, each one of which is supported in the back end by a corresponding parameter queue.

For our example OBSW model: `ParameterChannel` and `ParameterQueue` C++ classes as shown in Figure 6.11 are defined. These classes can be reused for any OBSW model i.e., they are generic code for all OBSW models.

All the infrastructure code entities mentioned above are present in the namespace `General` as these classes are used by both `Caller` and `Callee` components.

6.3.3.5. **Event emitter ports and event receiver ports**

The event emitter port for a particular event is mapped as an abstract base class and a corresponding concrete implementation class in C++. The event receiver port for a particular event is mapped only as an abstract base class. The abstract base class for event receiver port also contains pure virtual methods in order to safely interleave the reception of events.

For our example OBSW model: The `FailureEventEmitterPort` is mapped as a pair of abstract base class and a concrete implementation class as shown in Figure 6.12.

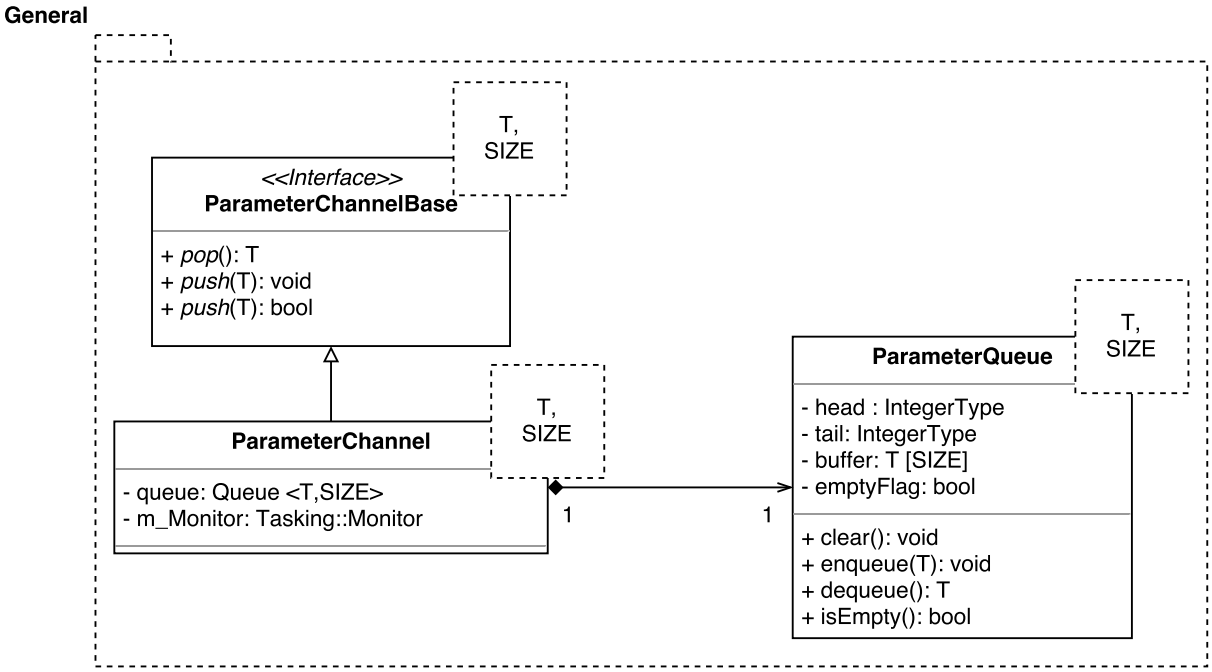


Figure 6.11.: UML class diagram representation of parameter channel and parameter queue in the example OBSW model

For our example OBSW model: The `FailureEventReceiverPort` is mapped to an abstract base class as shown in Figure 6.13.

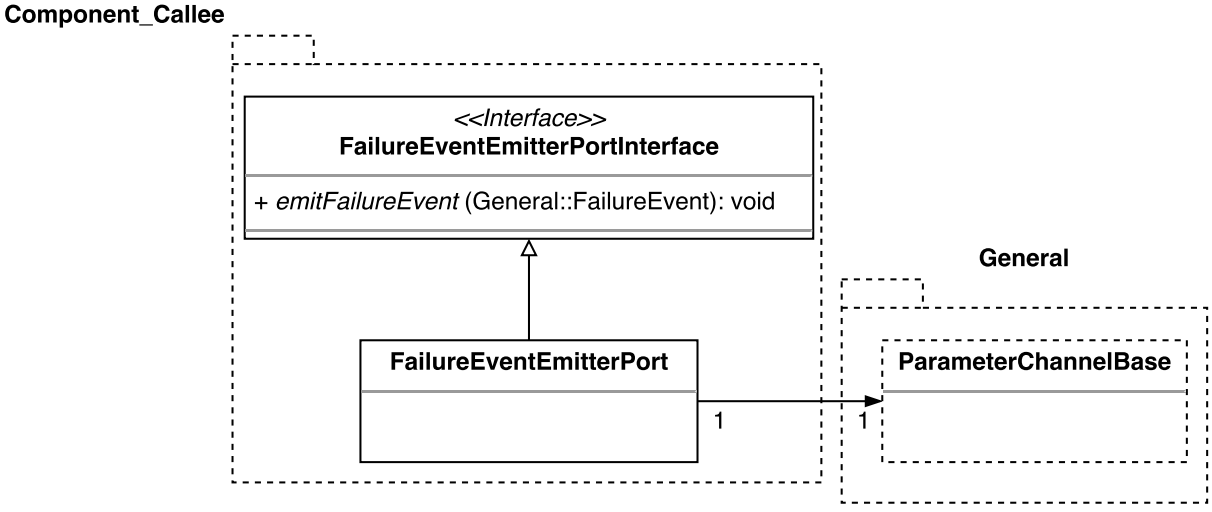


Figure 6.12.: UML class diagram representation of event emitter port in the example OBSW model

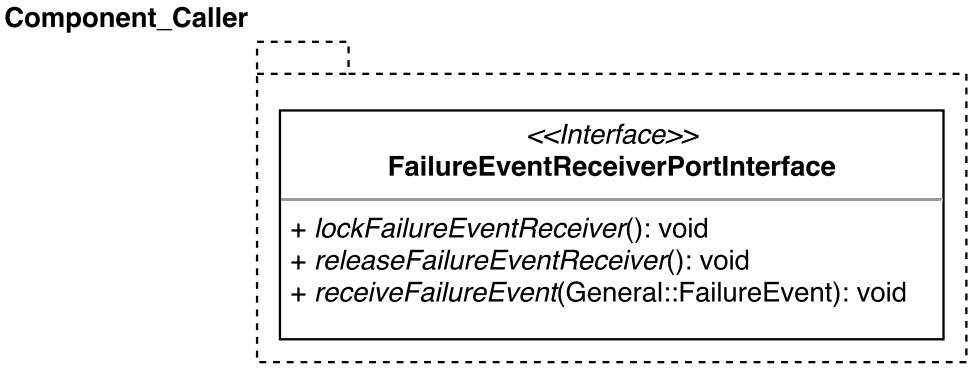


Figure 6.13.: UML class diagram representation of event receiver port in the example OBSW model

The `FailureEventEmitterPortInterface` and `FailureEventEmitterPort` classes are defined in the namespace `Component_Callee` as component `Callee` is the component which can emit a `FailureEvent`.

The `FailureEventReceiverPortInterface` is defined in the namespace `Component_Caller` as component `Caller` is the component which can receive a `FailureEvent`.

6.3.3.6. **Component types**

A component type can be mapped to an abstract base class in C++ as shown in [34]. A component type must provide all the operations that are listed in the provided interfaces of the component [33]. Hence it inherits from all the interface helper classes which are referenced by its provided interfaces as in [34].

This is where interface helper classes, with redefined operations come in handy, because C++ does not distinguish between operations with same signatures, although they are inherited from different namespaces. A component type must also inherit from the mapped abstract base classes for event receiver ports.

A component type must also have pure virtual methods which obtain and release the semaphores for the concurrent access of different operations that it provides. In addition to these, pure virtual methods need to be added, which act as call-back functions for the operations, that the component type's required interface ports request to be released asynchronously.

For our example OBSW model:

- As shown in Figure 6.14 `ComponentType` in the namespace `Component_Caller` inherits from the `InterfaceA_Helper` and also inherits from the abstract base class `FailureEventReceiverPortInterface`. It has pure virtual methods meant for:
  - Obtaining and releasing of semaphores for concurrent accesses of the operation `CallOperationAdd_InterfaceA`.
  - Call-back function for the operation `OperationAdd`.
  - Call-back function for the getter operation of the interface attribute `StatusValue`.
- A shown in Figure 6.15 `ComponentType` in the namespace `Component_Callee` inherits from the `InterfaceB_Helper`. It has pure virtual methods meant for:
  - Obtaining and releasing of semaphores for concurrent access of the operation OperationAdd_InterfaceB.
  - Obtaining and releasing of semaphores for concurrent access of the setter and getter operations for the interface attribute `StatusValue`.

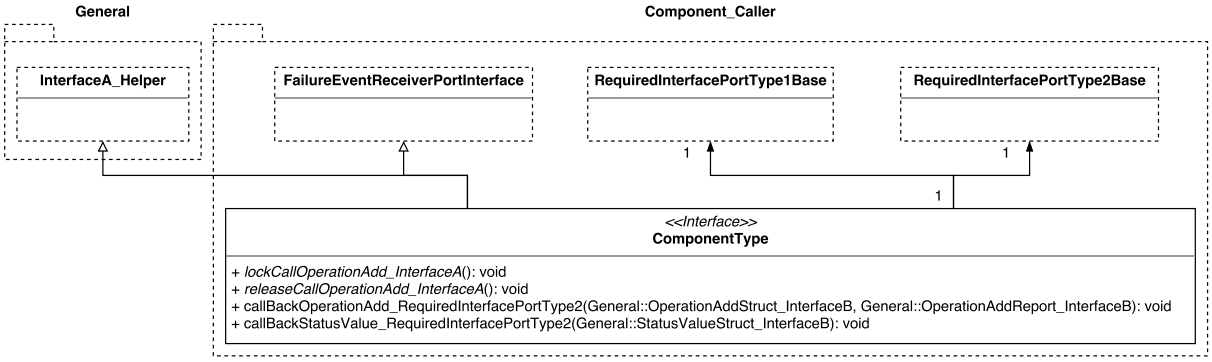


Figure 6.14.: UML class diagram representation of component type for `Component_Caller` in the example OBSW model

6.3.3.7. **Required interface ports**

A required interface port is mapped as an abstract base class and a corresponding concrete implementation class in C++ as shown in [34]. The required interface subsumed by a particular component type has various operations that it might request and each operation has information specified whether the required interaction kind is `synchronous` or `asynchronous` [32][33].

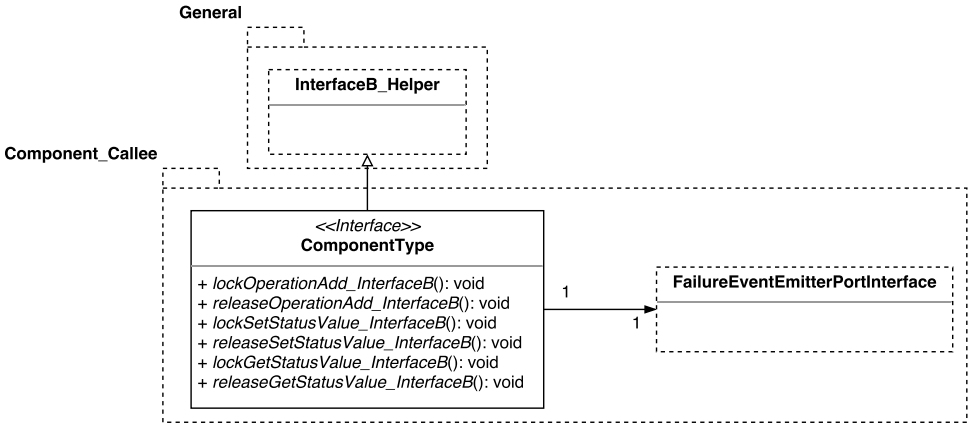


Figure 6.15.: UML class diagram representation of component type for `Component_Callee` in the example OBSW model

Each required interface port refers to one interface type and for each operation in the required interface port, a pure virtual method is added to the abstract base class. The signatures of these methods depend on whether the operations would be requested with an asynchronous release pattern or a synchronous release pattern.

In case of interface operations:

- If the desired interaction kind for the operation is `synchronous`, then the signature of the method in the abstract base class is same as the corresponding method in the abstract base class for the interface, as shown in Figure 6.16.
- If the desired interaction kind for the operation is `asynchronous`, then the signature of the method in the abstract base class bears no resemblance to the corresponding method in the abstract base class for the interface. It is changed as shown in Figure 6.17 to replace the `out` and `inout` parameters with a pointer to the desired call-back function.

In case of interface attributes:

- For the setter operation, signature of the method in the abstract class is same as the corresponding method in the abstract base class for the interface, as shown in Figure 6.16 and Figure 6.17.
- If the desired interaction kind for the getter operation is `synchronous`, then the signature of the method in the abstract class is same as the corresponding method in the abstract base class for the interface, as shown in Figure 6.16.
- If the desired interaction kind for the getter operation is `asynchronous`, then the signature of the method in the abstract base class bears no resemblance to the corresponding method in the abstract base class for the interface. It is changed as

Listing 6.2 Code excerpt from the generated code for requesting service `OperationAdd` in `RequiredInterfacePortType1`

```
General::OperationAddReport_InterfaceB RequiredInterfacePortType1::OperationAdd (const
    IntegerType& a,const IntegerType& b,IntegerType& c) {
      General::OperationAddReport_InterfaceB myReport;
      myReport = m_targetProvidedInterfacePort.OperationAdd(a,b,c); //Simple method call
      return myReport;
}
```

Listing 6.3 Code excerpt from the generated code for interface attribute `StatusValue` access in `RequiredInterfacePortType2`

```
void
    RequiredInterfacePortType2::getStatusValue(callBackStatusValue_RequiredInterfacePortType2
    callBackFunctionPtr) {
      StatusValueStruct_InterfaceB myStruct;
      myStruct.callBackFunctionPtr =
          std::bind(callBackFunctionPtr,m_myCallerInstance,std::placeholder::_1);
      m_targetProvidedInterfaceSlot.getStatusValue_Receiver(myStruct); //Simple method
          call
}
```

shown in Figure 6.17 to replace the method parameter with a single pointer to the desired call-back function.

The concrete implementation of any method, for example, as the one shown in the code excerpt in Listing 6.2 is fairly simple, if the desired interaction kind for the corresponding operation is `synchronous`. The implementation consists of a simple method call to the corresponding operation in the abstract base class of the bound provided interface port.

The concrete implementation for any method, for example, the one as shown in code excerpt in Listing 6.3 is fairly complex, if the desired interaction kind for the corresponding operation is `asynchronous`. It would do the following things as explained in [35]:

- Make local copies of all the method parameters (if any) as it is necessary to ask for an asynchronous release of the associated service.
- Pack them in the instances of data structures designed previously to carry the corresponding parameters.
- Simple method call to the corresponding operation in the abstract base class of the bound provided. interface port.

It is clear that local copies of the method parameters need to be made because the desired interaction kind is `asynchronous`.

For our example OBSW model: The following C++ classes are defined:

- `RequiredInterfacePortType1Base` abstract base class and its corresponding `RequiredInterfacePortType1` concrete implementation class.
- `RequiredInterfacePortType2Base` abstract base class and its corresponding `RequiredInterfacePortType2` concrete implementation class

The `RequiredInterfacePortType1Base` and `RequiredInterfacePortType2Base` have pure virtual methods as per the general description and they have to be implemented in the concrete implementation classes `RequiredInterfacePortType1` and `RequiredInterfacePortType2` respectively. The concrete implementations are in line with the description as in the general case above.

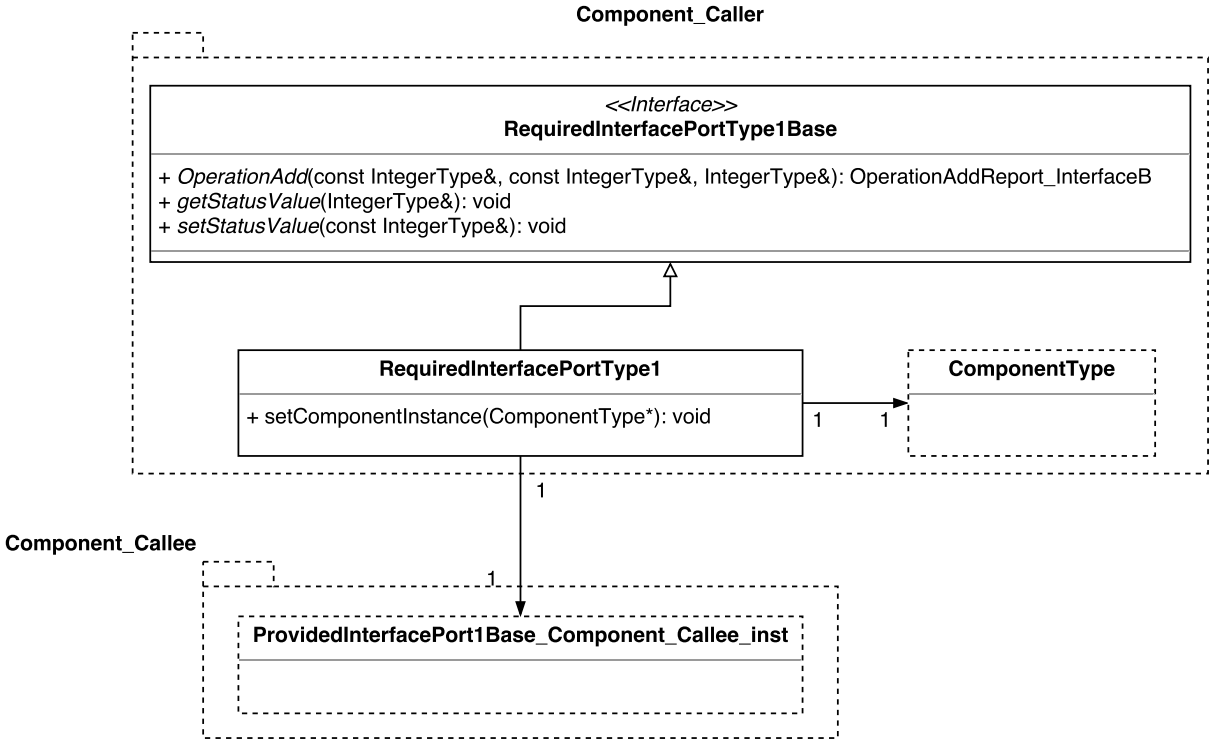


Figure 6.16.: UML class diagram representation of `RequiredInterfacePortType1` for `Component_Caller` in the example OBSW model

All the C++ classes mentioned above are present in the namespace `Component_Caller`. As the component type `Component_Callee` does not have any required interface ports, no C++ classes related to required interface ports are defined in the namespace `Component_Callee`.

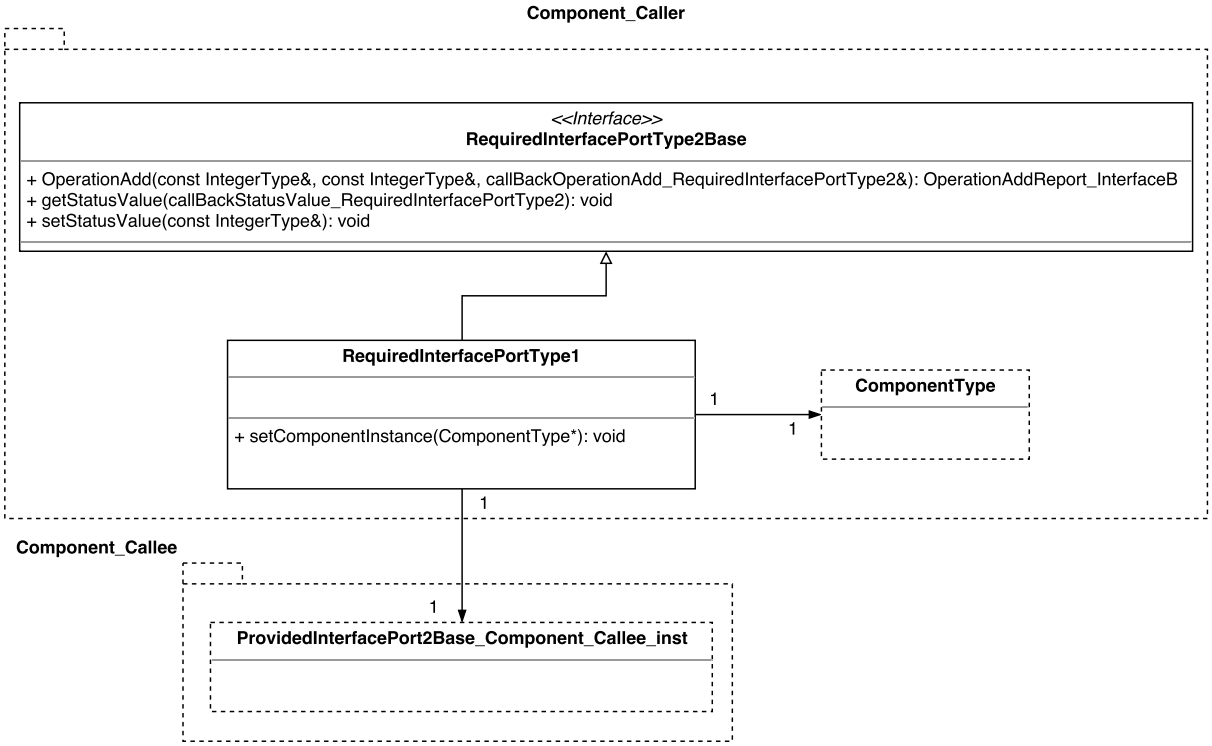


Figure 6.17.: UML class diagram representation of `RequiredInterfacePortType2` for `Component_Caller` in the example OBSW model

6.3.3.8. **Component implementations**

A component implementation can be mapped in C++ as a concrete implementation of its abstract component type base class. as in [34]. It implements all the pure virtual methods that are inherited from its component type. It also has actual instances of semaphores for allowing safe concurrent accesses to the implemented methods and for safe interleaving between concurrent receptions of events of the same type.

A variation of the very famous Template method design pattern as discussed in [**TemplatePattern**] is adopted in this Master thesis. It is used for the concrete implementation of the inherited operations. According to [**TemplatePattern**], it is better to hide the concrete implementation of the inherited virtual functions, and it is followed in this Master thesis.

In case of components that promote multiple provided interface ports which refer to the same interface type, it is necessary to provide multiple implementations for the operations in the provided interfaces. In order to solve this problem a class hierarchy as shown in Figure 6.19 is decided upon in this Master thesis. As this kind of class hierarchy involves multiple inheritance and is an eligible candidate for the Diamond Problem in

C++ [45], suitable measures are taken in the code generation step to avoid the usage of ambiguous base classes for polymorphic method calls:

- A component implementation abstract base class is designed which contains implementations for inherited pure virtual methods related to acquiring and releasing of semaphores. Also the inherited pure virtual methods which are provided only once are implemented here.
- A component implementation abstract base class is further extended by dummy abstract base classes. A dummy base class is added for each of the provided interface ports which refer to the same interface type. These dummy abstract base classes help in testing of the implementations of the services which are provided more than once by multiple provided interfaces. With the help of these dummy abstract base classes, mock implementation classes can be easily created and used for testing [12]. Also because the component type class for the intended component implementation classes with this kind of hierarchy, forms an ambiguous base, these dummy base classes are necessary for polymorphic method calls.
- These dummy abstract base classes are extended by concrete implementation classes which provide different concrete implementations for all the inherited operations except the ones, which are already implemented in the component implementation abstract base class. Template method pattern [**TemplatePattern**] is used in these concrete component implementation classes.
- As it is a necessity to have only one instantiable concrete implementation per component [34][32][33], all the concrete implementations are inherited one last time in a component implementation class. This instance is now deployable on the hardware platform.

For our example OBSW model:

- `ComponentImplementation` concrete implementation class in the namespace `Component_Caller` inherits from the abstract base class `Component_Type` in namespace `Component_Caller` as shown in Figure 6.18. It provides implementation for all the inherited pure virtual methods.
- Because the `ComponentType` in the namespace `Component_Callee` has two provided interface ports namely, `ProvidedInterfacePort1` and `ProvidedInterfacePort2`, which refer to `InterfaceB`, the following classes as shown in Figure 6.19 are created in the namespace `Component_Callee`:
  - `ComponentImplementationBase` which is an abstract base class, but provides implementation for inherited pure virtual methods related to acquiring and releasing of semaphores.

- `ComponentImplementationBase_ProvidedInterfacePort1` and `ComponentImplementationBase_ProvidedInterfacePort2` which are two dummy abstract base classes.
- `ComponentImplementation_ProvidedInterfacePort1` and `ComponentImplementation_ProvidedInterfacePort2` which provide different implementations for all the inherited operations except the ones implemented in `ComponentImplementationBase`.
- `ComponentImplementation` which inherits from both `ComponentImplementationProvidedInterface1` and `ComponentImplementationProvidedInterface2`.

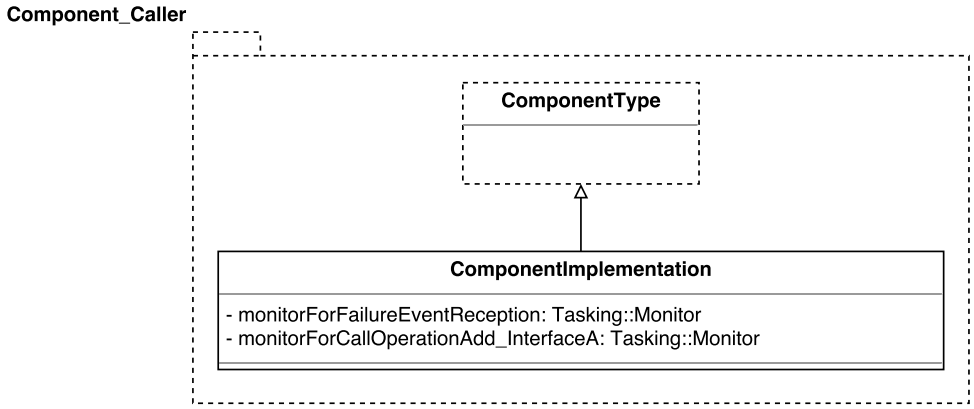


Figure 6.18.: UML class diagram representation of component implementation for `Component_Caller` in the example OBSW model

6.3.3.9. **Provided interface ports**

A provided interface port is mapped to an abstract base class and a corresponding concrete implementation class as shown in Figure 6.20 and Figure 6.21. The provided interface promoted by a particular component type has various operations that are provided and each operation has a desired release pattern attached as a non-functional/extra-functional property [32][33]. Each provided interface port refers to one interface type and for each operation in the provided interface port, a pure virtual method is added to the abstract base class. The signatures of these methods depend on whether these operations are requested with a synchronous release pattern or an asynchronous release pattern.

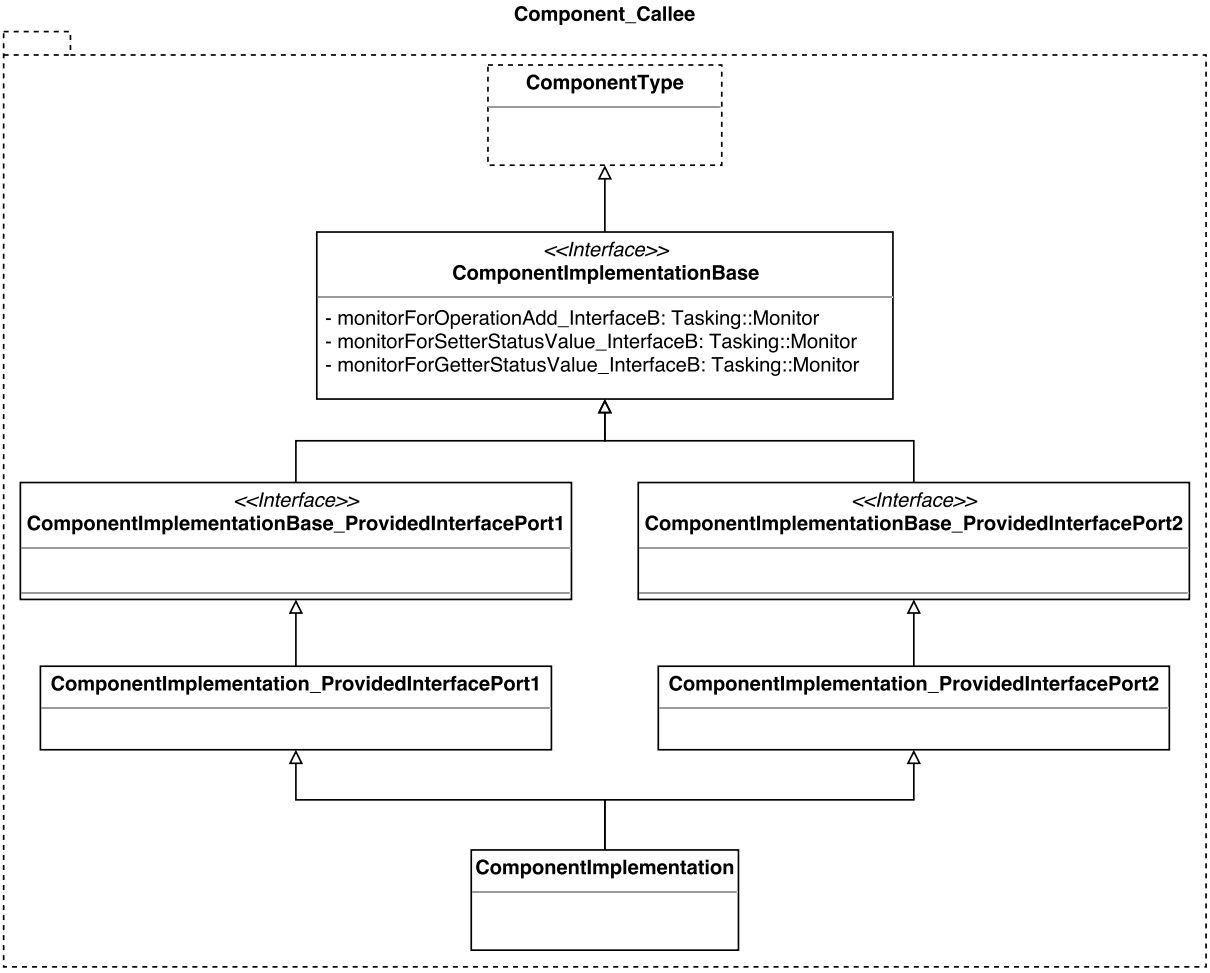


Figure 6.19.: UML class diagram representation of component implementations for `Component_Callee` in the example OBSW model

In case of interface operations:

- If the interface operation on the provided interface is expected to be called synchronously, then the signature of the method in the abstract base class is same as the corresponding method in the abstract base class for the interface, as shown in Figure 6.20 for `OperationAdd`.
- If the interface operation on the provided interface is expected to be called asynchronously, then the signature of the method in the abstract base class is changed as shown in Figure 6.21 for `OperationAdd`, to accept a data structure sent by the required interface port, designed to carry the values of the parameters for the operation along with a polymorphic function-wrapper for the call-back function.

In case of interface attributes:

- For the operations which set and get the values of the interface attributes synchronously, the signatures of the methods in the abstract base class are same as the corresponding method in the abstract base class for the interface.
- For the operations which set and get the values of the interface attributes asynchronously, the signatures of the methods in the abstract base class are changed to accept a data structure sent by the required interface port, designed to carry the values of the interface attribute, as shown in Figure 6.21. The data structure would also contain a valid polymorphic function-wrapper for the call-back function in case of getter operation for the interface attribute.

The following additional pure virtual methods are added to the abstract class for a provided interface port:

- For each interface operation and interface attribute setter/getter operation which is called asynchronously, an additional pure virtual method is added to store the address of the task channel, to which the data structure corresponding to the operation needs to be pushed. Refer figures Figure 5.2, Figure 5.3, Figure 5.4 in Section 5.2.2 of Chapter 5, where a diagrammatic representations of the required pushes onto the task channel are depicted.
- An additional pure virtual method for storing the reference to their corresponding component implementation base class as shown in Figure 6.20 and Figure 6.21. Using a reference to the component implementation, the service can be scheduled immediately in case of synchronous service release requests.
- An additional pure virtual method as shown in Figure 6.22, for storing the reference of the task from tasking framework which is responsible for periodically requesting the service in the provided interface port which has the interaction kind set to `cyclic`. Using this reference to the task, it is possible to ask the task to start its operation immediately or if necessary, after an initial offset [33].

The concrete implementation for the above mentioned pure-virtual methods, in case the corresponding operation is requested to be released synchronously, consists of a simple call to the corresponding method in the referenced component implementation base class as shown in code excerpt in Listing 6.4. If the non-functional property attached with release of the operation on the provided interface side is `Protected`, then the implementation also includes acquiring and releasing of semaphore associated with the operation as shown in code excerpt in Listing 6.4.

The concrete implementation for the above pure-virtual methods, in case the corresponding operation is requested to be released asynchronously, consist of pushing the data structure associated with the operation onto the corresponding task channel as

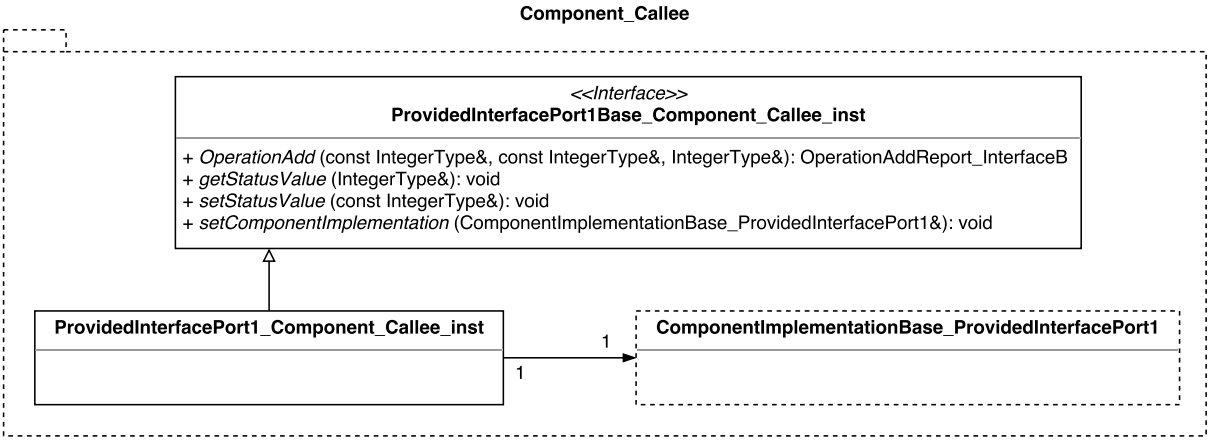


Figure 6.20.: UML class diagram representation of `ProvidedInterfacePort1_Component_Callee_inst` for `Component_Callee` in the example OBSW model

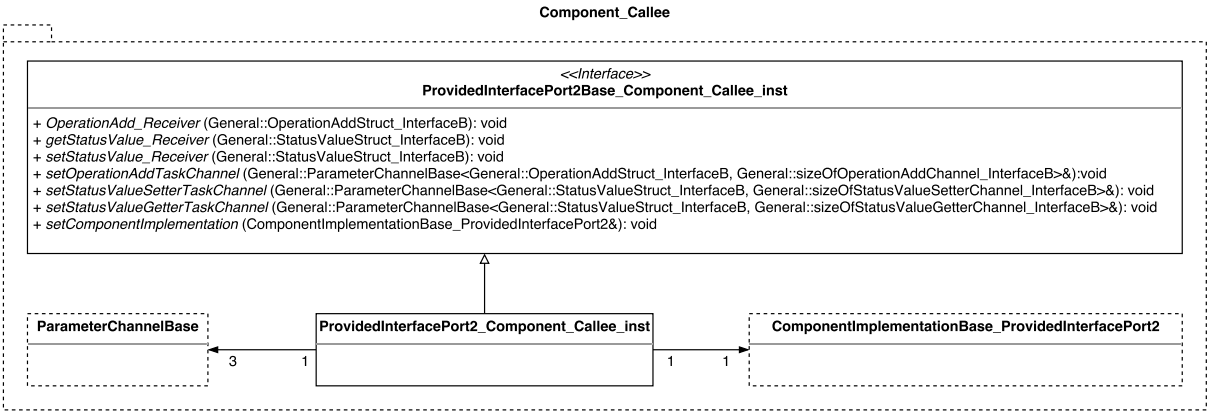


Figure 6.21.: UML class diagram representation of `ProvidedInterfacePort2_Component_Callee_inst` for `Component_Callee` in the example OBSW model

shown in code excerpt in Listing 6.5 . A diagrammatic representation of the push that is required, is already shown in figures Figure 5.2, Figure 5.3, Figure 5.4, in Section 5.2.2 of Chapter 5.

The concrete implementation for the above pure-virtual methods, in case the corresponding operation is requested to be released with non-functional property as `cyclic`, then it is a simple call to start the task using the reference to the task as shown in the code excerpt in Listing 6.6.

For our example OBSW model: The following classes as shown in Figure 6.20, Figure 6.21, Figure 6.22 are defined:

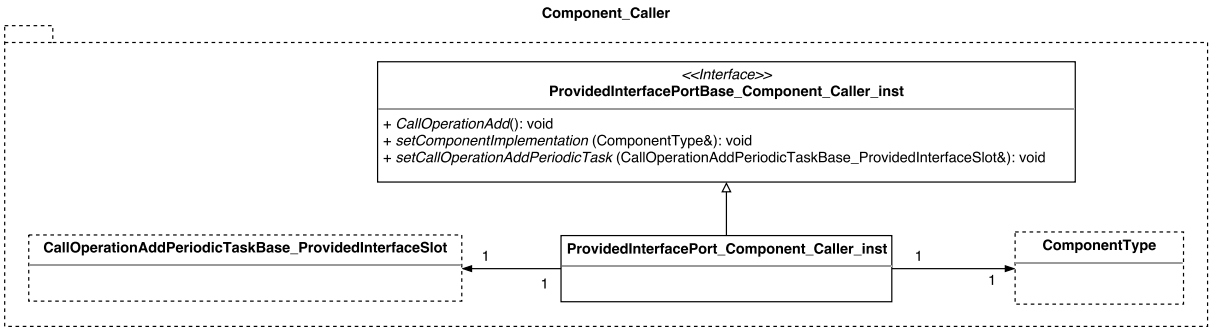


Figure 6.22.: UML class diagram representation of `ProvidedInterfacePort_Component_Caller_inst` for `Component_Caller` in the example OBSW model

Listing 6.4 Code excerpt from the generated code for operation `OperationAdd` access in `ProvidedInterfacePort1_Component_Callee_inst` which is called synchronously and has `Protected` as a non-functional property attached to it

```
General::OperationAddReport ProvidedInterfacePort1_Component_Callee_inst::OperationAdd
    (const IntegerType& a,const IntegerType& b,IntegerType& c) {
      General::OperationAddReport_InterfaceB myReport;
      m_myComponent->lockOperationAdd_InterfaceB();
      myReport = m_myComponent->OperationAdd(a,b,c); //Simple method call
      m_myComponent->releaseOperationAdd_InterfaceB();
      return myReport;
}
```

- `ProvidedInterfacePort1_BaseComponent_Callee_inst` abstract base class and its corresponding `ProvidedInterfacePort1_Component_Callee_inst` concrete implementation class.
- `ProvidedInterfacePort2_BaseComponent_Callee_inst` abstract base class and its corresponding `ProvidedInterfacePort2_Component_Callee_inst` concrete implementation class.
- `ProvidedInterfacePort_BaseComponent_Caller_inst` abstract base class and its corresponding `ProvidedInterfacePort_Component_Caller_inst` concrete implementation class.

Listing 6.5 Code excerpt from the generated code for operation `OperationAdd` access in `ProvidedInterfacePort2_Component_Callee_inst` which is called asynchronously

```
void ProvidedInterfacePort2_Component_Callee_inst::OperationAdd_Receiver
    (General::OperationAddStruct_InterfaceB myStruct) {
      m_myOperationAddChannel->push(myStruct); //A push onto the task channel
}
```

Listing 6.6 Code excerpt from the generated code for operation `CallOperationAdd` in `ProvidedInterfacePort_Component_Caller_inst` which has non-functional property set as `Cyclic`

```
void ProvidedInterfacePort_Component_Caller_inst::CallOperationAdd(void) {
      m_mycallOperationAddTask->startPeriodicTask(); //Method call to start the task
}
```

All the abstract base classes have pure virtual methods as per the general description given above and they have to be implemented in the corresponding concrete implementation classes. The concrete implementations are in line with the description as in the general case above.

6.3.3.10. **Tasks from the Tasking framework**

In the discussions about designing a programming model for OSRA in Chapter 5, it was clear that the threads of control might contain tasks from the Tasking framework. Each task is mapped to an abstract base class and a concrete implementation class in C++. A task would have instances of the required task inputs, task event as per the required threads of control explained Chapter 5.

Each task stores a reference to the base class of the component implementation in order to access the services implemented in the respective component implementations.

For our example OBSW model: The following classes as shown in Figure 6.23, Figure 6.24, Figure 6.25, Figure 6.26 are defined:

- `CallOperationAddPeriodicTaskBase_ProvidedInterfaceSlot` abstract base class and `CallOperationAddPeriodicTask_ProvidedInterfaceSlot` concrete implementation class, in order to call operation `CallOperationAdd` in the `ProvidedInterfaceSlot` perioidically every 2s.
- `OperationAddSporadicTaskBase_ProvidedInterfaceSlot2` abstract base class and a corresponding `OperationAddSporadicTask_ProvidedInterfaceSlot2` concrete implementation class, in order to call operation `OperationAdd` in the `ProvidedInterfaceSlot2` sporadically with a MIAT of 2s.
- `StatusValueSetterTaskBase_ProvidedInterfaceSlot2` abstract base class and a corresponding `StatusValueSetterTask_ProvidedInterfaceSlot2` concrete implementation class, in order to provide asynchronous access to the setter operation of the interface attribute `StatusValue` in the `ProvidedInterfaceSlot2`.

- `StatusValueGetterTaskBase_ProvidedInterfaceSlot2` abstract base class and a corresponding `StatusValueGetterTask_ProvidedInterfaceSlot2` concrete implementation class, in order to provide asynchronous access to the getter operation of the interface attribute `StatusValue` in the `ProvidedInterfaceSlot2`.
- `FailureEventReceiverTaskBase` abstract base class and a corresponding `Failure EventReceiverTask` concrete implementation class, for the reception of `Failure Event` which are emitted by the `Component_Callee` asynchronously and to forward it to `Component_Caller`.

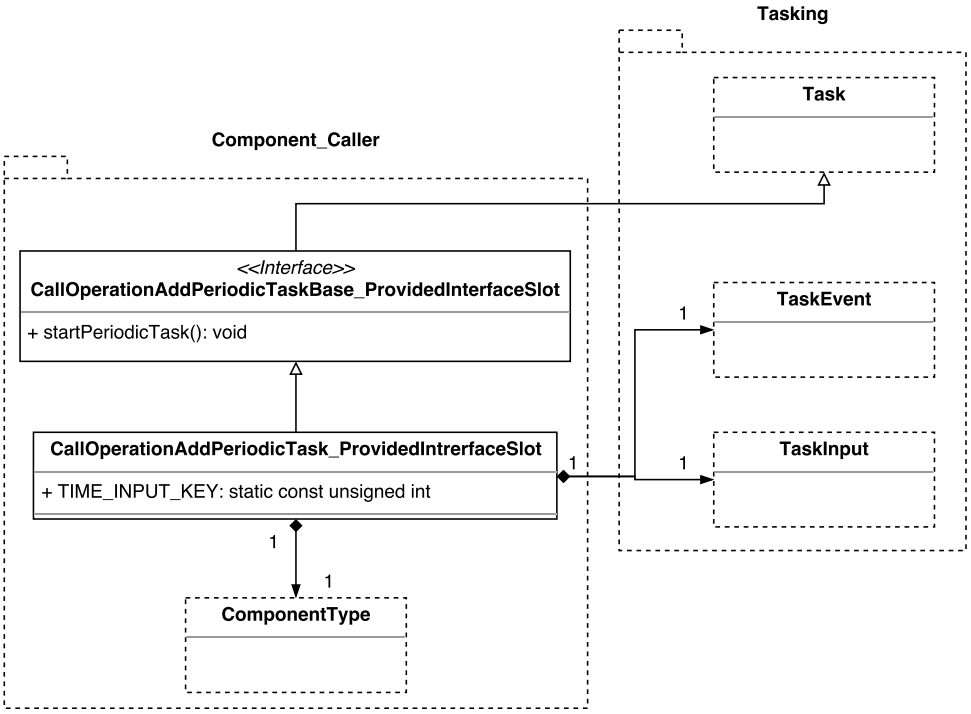


Figure 6.23.: UML class diagram representation of the task required to call `CallOperationAdd` in `ProvidedInterfacePort_Component_Caller_inst` perioidcally with a period of 2*s* in the example OBSW model

6.3.3.11. **Component containers**

A container of a component can be mapped to a class in C++ as in [34]. A component container consists of the following entities:

- Instances of required interface ports.
- Instance of component implementation.

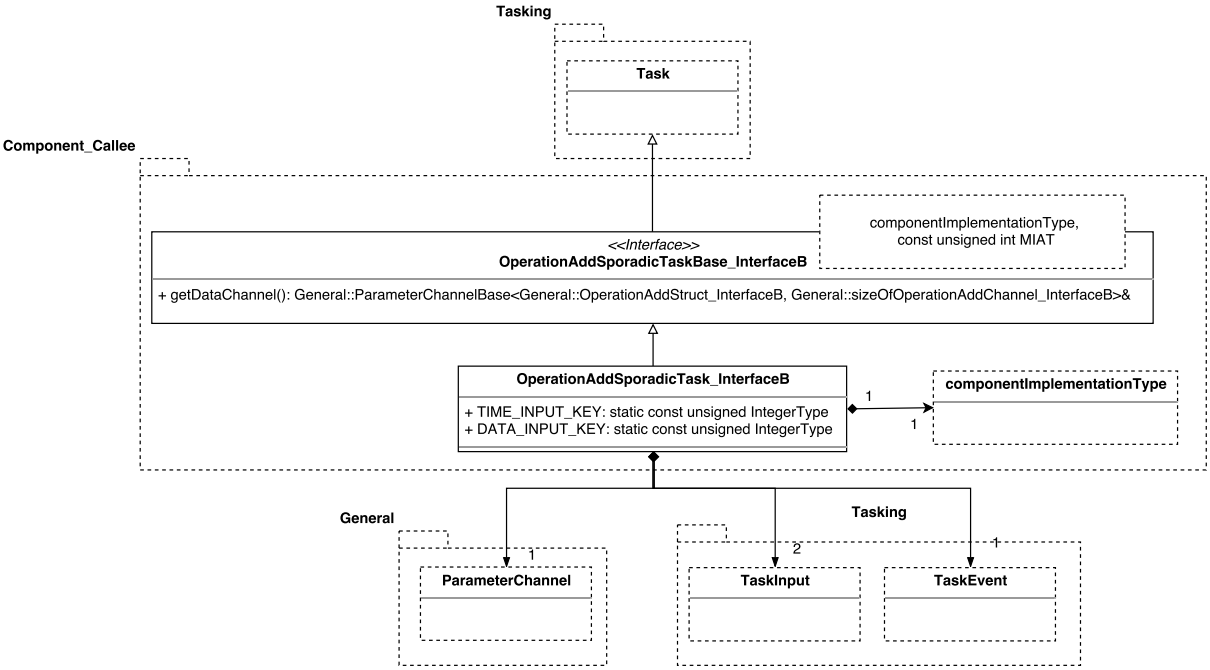


Figure 6.24.: UML class diagram representation of the task required to call `OperationAdd` in `ProvidedInterfacePort2_Component_Callee_inst` sporadically with a MIAT of 2*s* in the example OBSW model

- Instances of tasks which are necessary to handle services which are called asynchronously.
- Instances of tasks which are necessary to receive asynchronous events.
- Instances of event emitter ports.

A component container is also responsible for initializing the following:

- The instantiated component instance by providing references of the event emitter ports.
- The event emitter ports by providing reference of the corresponding task channel to which a emitter port needs to push an instance of the event.
- The provided interface ports by:
  - Triggering the operations in the provided interface ports which have the desired non-function property set as `cyclic`.
  - Providing reference of the component instance in order to access the service directly in case the service is requested to be released synchronously.

  - Providing references of the different task channels they need to push the data structures associated with the operations onto, in order to handle the services which are requested to be released asynchronously.

- The instances of tasks with reference of component instance in order to schedule the execution of the services.
- The required interface ports by providing references of:
  - The instantiated component instance, in order to initialize the data structures of the operations with correct function wrappers for call-back functions.
  - The respective provided interface port each one of them is bound to.
- The event receiver tasks with reference of the corresponding component instance and the corresponding channels which needs to be associated with their respective task inputs.

For our example OBSW model: The following classes as shown in Figure 6.27 and Figure 6.28 are defined:

- `Container` in the namespace `Component_Caller` which is the container for the component instance `Component_Caller_inst` and its provided and required interface slots.
- `Container` in the namespace `Component_Callee` which is the container for the component instance `Component_Callee_inst` and its provided interface slots.

Both the containers have instances of their respective different components as explained in the general case and as shown in Figure 6.27 and Figure 6.28. They are responsible for the initialization of different components as explained in the above general description.

6.4. Code generation using Xtend

The reference implementation of the OSRA component model consists in a set of .ecore metamodels [32]. Ecore is an implementation of the Essential Meta Object Facility (EMOF), the meta-meta language by the OMG for the specification of meta-models [32]. The main advantage of using Ecore is the ready availability of graphical editors for the specification of metamodels and powerful support provided by the Eclipse Modeling Framework (EMF), which is a framework of the Eclipse development platform that permits to generate a code implementation of the metamodel entities, basic editors for the creation of models conforming to the metamodel under development [32].

It is an implementation decision in this Master thesis to use Xtend for the code generation. Xtend is a general purpose Java-like language that is completely operable with Java [2][47]. Xtend has a more concise syntax than Java and provides powerful features such as type inference, extension methods, dispatch methods, an lambda expressions and the all important multiline template expressions, which are useful when writing code generators [2][47]. Xtend also provides powerful features that make model visiting and traversing really easy, straightforward, and natural to read and maintain.

Xtext is an eclipse framework for implementing programming languages and Domain-Specific language (DSL) [2]. Xtext helps to implement languages quickly, and most of all, it covers all the aspects of a complete language infrastructure like parser, code generator etc. Xtext uses Google Guice, which is a dependency injection framework to create and call a code generator [2]. The dependency injection pattern basically allows to inject implementation objects into a class hierarchy in a consistent way [12]. The Xtext's generator support can be used in Xtend directly to build code generators for non-Xtext based models, such as the OBSW models constructed using the OSRA component model [9].

The tutorial in [44] is used as a base in this Master thesis to construct a code generator using Xtend for non-Xtext based models and also provide a UI integration for the code generator.

6.5. Organizing the generated code

As explained in the previous sections, adopting separation of concerns even at the implementation level is one of the primary goals of this chapter and we have successfully achieved it in the discussions on software design for the infrastructural code in Section 6.3.3.

To further emphasize on separation of concerns at the implementation level, it is necessary to properly separate the generated infrastructure code into a meaningful files and a suitable file structure helping the third party software supplier to separate the automatically generated code from the code that needs to be supplied. This is of prime importance for the third party software supplier to not accidentally lose the implementations in successive code generation cycles.

The overall idea would be to:

- Generate two types of folders to clearly separate the infrastructural code entities, at the component type level from the infrastructural code entities at the component

instance level. The number of folders at the component instance level, depends on the actual number of component instances.

  - The first folder type would hold C++ classes related to its component type, required interface ports, event emitter ports, event receiver ports in the sub-folder named as `AutogeneratedCode`. It also contains C++ classes related to component implementations and because it is the unit of sub-contract, it is placed in a separate sub-folder named as `UserCode`. The third-party software supplier can alter the code in this folder safely without the fear of the code being overwritten by successive code generation cycles. Care is taken in the code generator to generate the classes in these files only once
  - The second folder type would hold the C++ classes related to its component instances, namely, provided interface ports, component containers in the sub-folder named as `AutogeneratedCode`. As already explained the component container class for a component instance would contain provided and required interface slots, component instance itself

- The folder named as `DatatypesInterfacesEventsAndExceptions` would hold C++ classes related to the data types, exceptions, events, parameter channel and their corresponding parameter queues

The folder structure for our running example is listed in Appendix A.

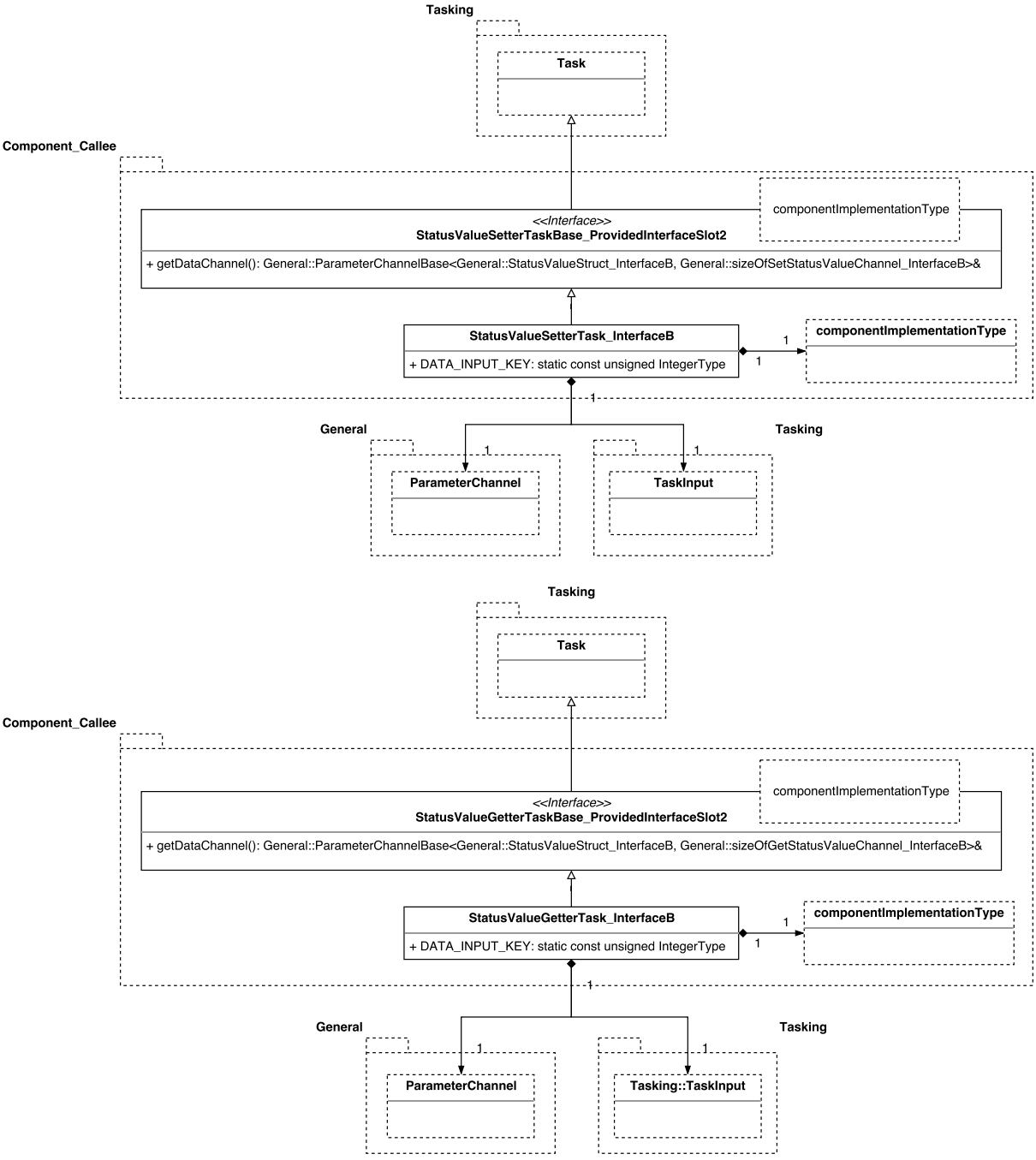


Figure 6.25.: UML class diagram representation of the tasks required to set and get the values of the interface attributes asynchronously in `ProvidedInterface Port2` in the example OBSW model

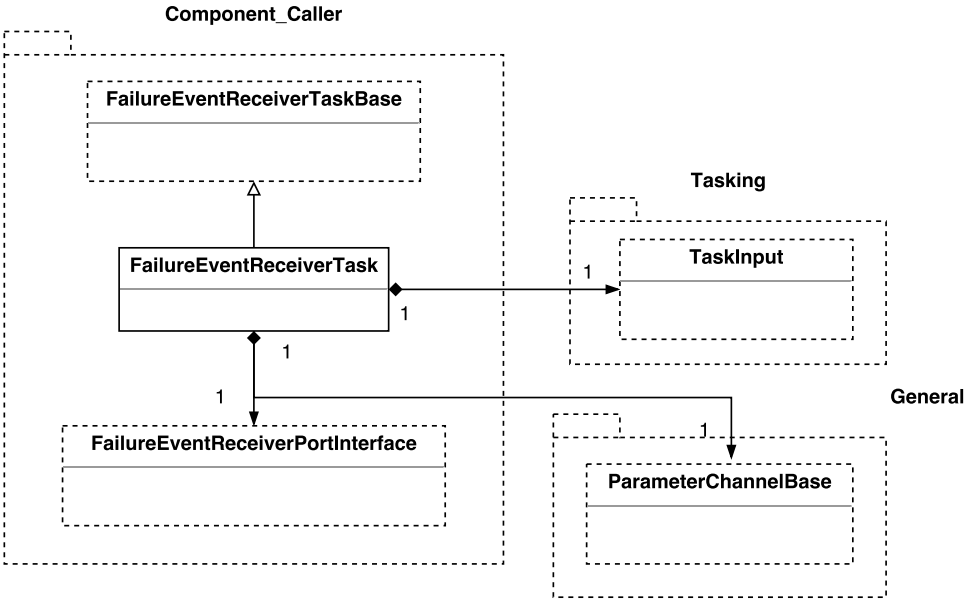


Figure 6.26.: UML class diagram representation of the task required for the reception of the `FailureEvent` in the example OBSW model

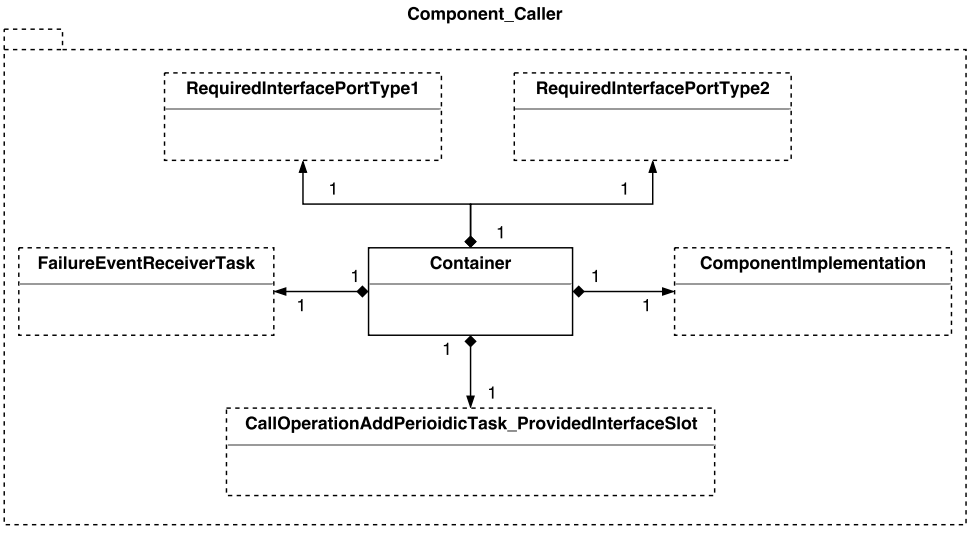


Figure 6.27.: UML class diagram representation of the container for `Component_Caller` in the example OBSW model

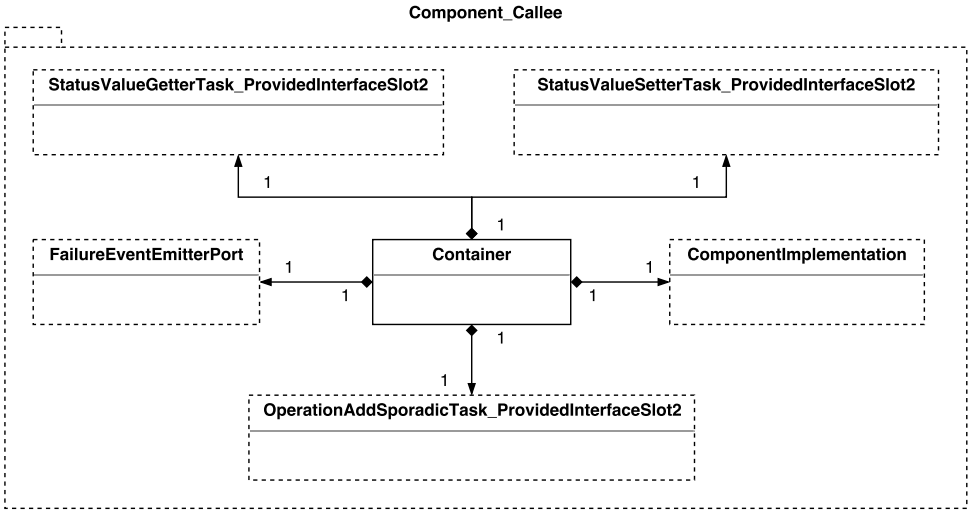


Figure 6.28.: UML class diagram representation of the container for `Component_Callee` in the example OBSW model

Chapter 7

Evaluation of the code generator

7.1. Introduction

By using model-driven engineering tools (MDE) tools for code generation, it is possible to generate software code automatically and achieve extremely high developer productivity rates of thousands of function points and millions of lines of code per person-month [19]. But, as we have seen in the previous chapters, the MDE approach consists of more than code generation tools; It defines the entire software-engineering approach that can impact the entire lifecycle from requirements gathering through sustainment [33][1].

It is important to consider these tools and methods in the context of a particular system acquisition i.e., the MDE methods and tools need to be aligned with the system acquisition strategies, which would in turn improve system quality, reduce time to field, and reduce sustainment cost [19]. System acquisition strategies include:

- Securing the necessary data rights and licensing for tools, models, generated code, run-time libraries, frameworks, and other supporting software
- Reviewing and evaluating appropriate artifacts introduced by the MDE tools at the right time in the acquisition cycle
- Approaches to manage program risks, include risk identification and mitigation

If the methods and tools do not align with the system acquisition strategy, using them can result in increased risk and cost in development and sustainment[19]. The acquirers in government or large commercial enterprises have the challenge of selecting contractors to develop their systems. The tools and processes selected by the contractors and developers have direct impact on the software quality concerns of the acquirer, who often has little influence on the selection of these tools and processes. The tool acquirer would then have to answer the following acquisition evaluation questions [19]:

- Do the engineering processes and associated development tools match the desired acquisition strategy?
- Do the tools support the developer's software development methodology?
- Are the code generation tools capable of integrating with other development and management tools to support measurement and monitoring of the development progress?
- Will the selected development methodology with its associated tools be available and compatible for the expected lifecycle of the system?

This chapter subjects the code generator developed as a part of the Master thesis to evaluation methods listed in [19] and provide necessary inputs for conducting the acquisition evaluation.

7.2. Selection and evaluation methods of a MDE tool for code generation

The step-by-step MDE tool selection process defined in [19] makes use of the Plan Establish Collect Analyze (PECA) method [7] as shown in Figure 7.1.

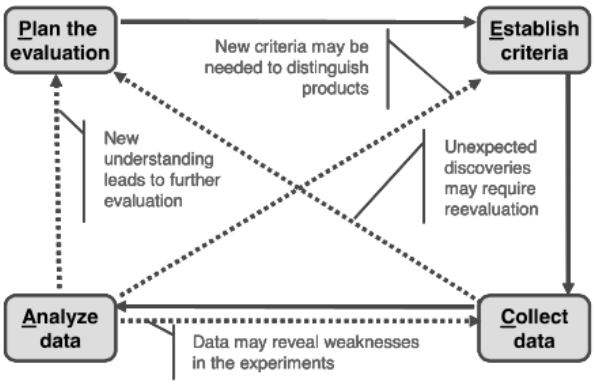


Figure 7.1.: The PECA Process

Source: [7]

As a part of the `Establish criteria` step in the PECA process, the acquirer must establish criteria with which he has to decide whether a particular tool for automatic code generation is suitable for a specific system acquisition. Such a criteria can be developed using risk taxonomy [8] which ensures that all relevant acquisitions strategies

are covered i.e., it provides a checklist to ensure all potential risks are considered [19]. The risk taxonomy has three main sections [8]:

Product engineering This covers activities that create a system that satisfies the specified requirements and customer expectations. Risks in this area generally arise from requirements that arise from requirements that are technically difficult to achieve, inadequate requirements and design analysis, or poor design and implementation quality

Development environment This includes risks related to the development process and system, management methods, and work environment

Program constraints This cover risks that arise from factors external to the project

To establish a criteria for a particular program or project, it is necessary for the program to first scan the risk taxonomy and identify those areas that apply for the project. Each risk creates one or more acquisition concerns , which may refine the program risk or indicate how certain tool features or capabilities might help mitigate the risk [19].

As part of the `Collect data` step in the PECA process, a vendor self-assessment questionnaire is prepared as in [19] which is given to the MDE tool vendors, to provide data needed to make the tool selection decision.

As a part of the `Analyze data` in the PECA rocess, it is then necessary to position these acquisition concerns in the specific program context and finally decide whether acquiring a particular tool is beneficial for the project.

In this Master thesis, a subset of evaluation criteria is chosen from Appendix A in [19]. The risk areas and the acquisition concerns which are meaningful in the scope of this Master thesis are chosen. For example, the [19] discusses about potential risks in the Process Management area of the project and these are of no interest in this Master thesis and are not considered. Each of the acquisition concerns in Appendix A of [19] are then linked to the questions in the vendor self-assessment questionnaire listed in Appendix B of the [19]. For our chosen subset of potential areas and the acquisition concerns within this Master thesis, an attempt is made to answer the corresponding linked questions in the questionnaire. The tables Table 7.1, Table 7.2, Table 7.3, Table 7.4, Table 7.5, Table 7.6 showcase these efforts.

Table 7.1.: Evaluation Criteria - Product Engineering Risk Area (Requirements)

Risk area	**Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation**	**Answers to the linked questions in the questionnaire**
Requirements		
Stability	Responding to requirements changes may necessitate operating on partially complete models and performing refactoring or rework on models.	The OBSW models designed using the OSRA Component Model should be subjected to model validation against the OSRA Specification Compliance and the SCM Metamodel Compliance before it is subjected to automatic code generation. In that case, the OBSW model, even if partially complete can be subjected to code generation if it clears the model validation step
	Communication with stakeholders is partially important to resolve requirements issues, so tool features that support this become more important	It is possible to annotate each of the model entities while constructing the OBSW model with information which can be used for communicating with the stakeholders. There is no additional documentation tool which comes with the OSRA SCM tool suite
	Interfaces between the software modeling tools and the requirements management tools promote co-evolution of requirements and software	There is no support for tracing requirements into model elements at the current state of development of OSRA SCM
Completeness Clarity Validity	In addition to the concerns noted above about requirements stability, the ability to execute or simulate the execution of the model can help validate requirements completeness	At the current stage of the development of the OSRA SCM, it is not possible visualize the execution of the model. It is only possible to create static models of the OBSW and it is not possible to trace the flow of execution through the model, or inject data or events into the model
Feasibility	The ability to perform analysis of the model for qualities such as latency, throughput and consistency can help demonstrate the feasibility of requirements	The model-based static analysis of the OBSW model is not possible at the current stage of development of the OSRA. Step 10 of the overall software development process in Section 3.2 in chapter Chapter 3 gives an idea about the analysis of latency, throughput etc. which can be performed on the OBSW model in the future
Scale	Limitation on the size or complexity of the model that can be represented, analyzed, or transformed by the tool will limit the scale of the system that can be created	There are no size and complexity limitations for representing the OBSW models using the OSRA SCM tools

Table 7.2.: Evaluation Criteria - Product Engineering Risk Area (Design)

Risk area	Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation	Answers to the linked questions in the questionnaire
Design		
Interfaces	If only parts of the system will be automatically generated, while other parts will be developed using traditional approaches, the interfaces between these two types of software must be designed and developed	In the software design for the generated infrastructure code, the model entities are mapped to infrastructural code entities. As the model entities clearly separates the two types of code, the generated infrastructural code entities as well clearly separates the generated code from the user code, which may be developed using traditional approaches. The generated source code needs to be compiled. The generated C++ source code works with the Tasking framework written in C++. The code is generated for the Linux platform and GCC C++ compiler conforming to the C++11 standard can be used to compile to the source code. The code generator also generates SCons build scripts for building the generated software code automatically
Testability	The tool should generate code that exposes internal states and interfaces needed to test the generated software	The generated infrastructural code is testable as testability of the generated code is one of the main concerns in the software design for the infrastructural code. (Effective testability of the generated code is proven with an example in the subsequent chapter) Mock C++ classes for different infrastructural code entities and automatic test cases can also be generated in future using the code generator
Hardware constraints	The generated code must be sufficiently efficient (in terms of processors, memory, network, disk, and other resource utilization) to operate in the target environment	The generated code is based on Tasking framework written in C++. The generated code at the moment is not optimized for efficient usage of processor and memory, however they can be targeted in an extension to this Master thesis. The Tasking framework plays an important role in effective utilization and abstractization of the hardware resources, resource handling in the Linux environment
Non-developmental software	Any runtime packages, libraries, or other software required to execute the generated code must be known, compatible with the current and future target environments, and able to be certified for use in other environments	The generated code depends on Tasking framework which is written in C++. The generated code uses Linux as the target environment and hence uses Tasking framework which sits on top of the Linux POSIX library. As Tasking framework is internal to DLR, no certification efforts have been done. The compatibility of Tasking framework for future environments is not of concern in this Master thesis. The build system which is automatically generated by the code generator makes use of SCons for building the generated code

Table 7.3.: Evaluation Criteria - Product Engineering Risk Area (Integration and Test)

Risk area	Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation	Answers to the linked questions in the questionnaire
Integration and Test		
Environment	The generated code may have run-time dependencies on third party software or software provided by the tool vendor. This software must be compatible with the integration environment	The generated code has run-time dependencies on the Tasking framework which is internal to DLR and no integration tests are automatically generated at the moment from the code generator SCons build scripts are also automatically generated by the code generator which helps in building the generated code
Product System	The tool should generate code that exposes internal states and interfaces needed to test the generated software code	The generated infrastructural code is testable as testability of the generated code is one of the main concerns in the software design for the infrastructural code. (Effective test -ability of the generated code is proven with an example in the subsequent chapter) Mock C++ classes for different infrastructural code entities and automatic test cases can also be generated in future using the code generator

Table 7.4.: Evaluation Criteria - Program Constraints Risk Area (Resources)

Risk area	Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation	Answers to the linked questions in the questionnaire
Resources		
Schedule	Developer productivity is measured differently when using automatic code generation approaches	There is definite increase in the developer productivity because all the infrastructural code are automatically generated and the third-party software supplier needs to concentrate only on the functional code which is most of the times sequential in nature [\{]CompBasedProcess}. No metrics are however available from completed project
Budget	The tool, including any optional features (eg. import, export and integration with other tools) along with the environment to execute the tool, must be acquired	As the generated code makes use of the Tasking framework written in C++ and the target platform for the generated code is Linux, it is necessary for the organization to use this platform. As the code generator is itself an Eclipse plugin, it is necessary for the organization to acquire Eclipse tool suite for Java and DSL developers. If necessary the organization may also acquire the OSRA SCM model editor as well. Also, SCons software construction tool is necessary to use the build scripts that are generated for building the generated code. GCC C++-11 compatible compiler is required for compiling the generated software code These tools need to be acquired and can cause variation in the budget for the project

Table 7.5.: Evaluation Criteria - Development Environment Risk Area (Development Process)

Risk area	**Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation**	**Answers to the linked questions in the questionnaire**
Development Process		
Process Control	The tool must provide mechanisms for maintaining consistency of modeling, analysis, and generation at the scale required to develop and sustain the system (eg. multiple teams, multiple connected sites, and multiple contractors)	It is unclear whether the overall software development process as a result of adopting OSRA SCM and its tool suite, is scalable for multiple teams, multiple connected sites, and multiple contractors. The code generator which is developed as a part of this Master thesis does not support them as well It is possible for the user to define reusable templates (eg. Interfaces) that the user can later adopt in another model
Product Control	An update to the tool may necessitate repeating model analyses, repeating code generation, and repeating test, integration, and certification. Tools that are rapidly evolving may put a strain on the development process	Any update to the code generation tool which is developed as part of this Master thesis, necessitates repeating code generation and all the efforts such as repeating test cases generation, integration and certification that follows the code generation
	If the tool is delivered as a service, then configuration control of the tool is provided by the vendor	At the current stage of development, the OSRA tool suite does not include a configuration control tool

Table 7.6.: Evaluation Criteria - Development Environment Risk Area (Development System)

Risk area	Potential Acquisition Concerns Related to MDE Tools for Automatic Code Generation	Answers to the linked questions in the questionnaire
Development System		
Capacity	The tool's modeling, analysis, and code generation environment will require the development and sustainment organization to deploy particular platforms and prerequisite software	As the generated code makes use of the Tasking framework written in C++ and the target platform for the generated code is Linux, it is necessary for the organization to use this platform. As the code generator is itself an Eclipse plugin,it is necessary for the organization to acquire Eclipse tool suite for Java and DSL developers. If necessary the organization may also acquire the OSRA SCM model editor as well. Also, SCons software construction tool is necessary to use the build scripts that are generated for building the generated code. GCC C++-11 compatible compiler is required for compiling the generated software code.
Usability	Integration of the tool with upstream (e.g., requirements mgt.) and downstream (e.g.,integration or certification) tooling improves usability	At the current phase of development of OSRA tool suite there are no tools which help in requirements management, integration or certification of the generated code
Familiarity	If the development and sustainment teams are not familiar with the tool, training and support must be available to enable the organizations to gain the knowledge and skills needed to develop and sustain the software	As the OSRA software development process makes use of the Component-Based Software Development process and Model Based Software Engineering, it is necessary that the development and sustainment teams need to have prior knowledge about these processes. The development teams also need time to have a good understanding of the OSRA SCM component model to effectively design the OBSW models. They also need to be trained to use the OSRA SCM model editor and the code generator which is designed as a part of this Master thesis

Chapter 8

Results and Conclusions

8.1. Discussion

As a part of this Master thesis, the following

- A choice is made to use the Tasking framework as a computational model so that the OSRA component model statically binds to the Tasking framework which formally defines the computational entities and the rules which govern their usage
- A reference programming model is decided upon that enforces the analysis assumptions and which permits to express exclusively the semantics imposed by the analysis theory and which conveys the implementations of the desired non-functional properties using the primitives from the Tasking framework
- Different corner cases which might arise during the construction of an OBSW model using the OSRA component model are identified
- An overall software design approach for the generated infrastructure code of the OBSW models is presented and a mapping of the OBSW model design entities to the infrastructural code entities is presented.
- A code generator is implemented, using which the generation of the entire non-functional code i.e., the code for handling the concurrency and interaction requirements for communication between components and generation of component containers and component connectors can be automated. The code generator uses the already tried and tested Tasking framework as the platform and bases the generated code on it. The advantage of this is that it eases the model-to-code transformation step
- The implemented code generator is tested for multiple OBSW models as shown in Appendix B which capture the different corner cases identified

- For the simple OBSW model example which was introduced in Chapter 6, a set of unit test cases are written using Gtest and Gmock frameworks and the test coverage reports are generated

The following results were obtained

- The implemented code generator successfully generates the infrastructural code entities for all the example OBSW models listed in Chapter 6 and in Appendix B. The generated code in all cases is successfully compiled to an executable along with Tasking framework using GCC C++ compiler conforming to the C++11 standard for the Linux platform
- The test coverage reports generated for the unit tests written for the simple OBSW example are analyzed. The results show that the testability factor of the generated code is high and the infrastructural code entities can be efficiently tested, meeting the needs for high testability of the generated software

8.2. Identified shortcomings of Tasking framework

During the course of the Master thesis, the following shortcomings of the current version of Tasking framework, which is chosen as a computational model for this Master thesis are identified:

- Tasks from Tasking framework are used in various threads of control as explained in Chapter 5. At the heart of the Tasking framework is a scheduler which schedules tasks based on priorities and these tasks are non-preemptible at the moment [22]. This is one of the critical shortcomings in the current version of the Tasking framework as it makes the generated software code which is based on Tasking framework not suitable for hard real-time systems [25]. Time-monitoring architectures such as Server-based architecture or Priority-Band architectures listed in [25] need to be adopted in the Tasking framework in order to make the generated code truly real-time capable. These architectures help in providing isolation of applications i.e., tasks (at least) along three orthogonal dimensional axes: time, space and communication [25]
- In the current version of the Tasking framework there is no possibility to measure the run-time of the tasks and monitor deadline violations which are mostly caused by WCET overruns of either the task at hand or a higher priority task. This limits the extent of property preservation in the model-to-code transformation step [25]. It is of very high importance that the system properties asserted during the

analysis and the assumptions made for the analysis to hold are preserved across implementation and execution [34][25]

- It is also not possible to measure the execution time of a group of tasks which are associated to the single time budget so that the group budget is accounted for their collective execution time. This incapability makes the adoption of Server-based architecture in the Tasking framework more difficult [25]
- In line with the inability to measure the run-time of the tasks from the Tasking framework, Tasking Framework also does not provide any constructs for at least coarse-grained fault detection and fault handling in case of deadline misses

8.3. Future Work

As an enhancement to the current work, it is possible to extend this Master thesis in the following directions:

- A rather straight forward improvement would be to generate the unit test cases and mock classes for different infrastructural code entities automatically as part of the model-to-code transformation step. This helps in automating the testing of the generated code and testing the code entities independent of each other
- The generated infrastructure code is found to have all the good characteristics of a software as listed in Section 6.3.3 although the generated code needs to be rigorously evaluated for each of these characteristics
- Enable model-based round-trip analysis by conducting static analysis of the system model in the non-functional dimensions of interest. For example, schedulability analysis which verifies whether the timing non-functional requirements can be met [3],[33]. For this, a platform specific model (PSM) i.e., a Schedulability Analysis Model (SAM) can be created from the declarative specification of the concurrent semantics that decorate the user model and inputs for the schedulability analysis tools can be automated and the results of the analysis can be seamlessly propogated back to the user space [3]
- Model-to-code transformation of more complex data types such as opaque types, arrays, structures, unions etc which are possible to be instantiated in the OBSW model
- Modeling of the relevant aspects of the hardware architecture and of the execution platform services are not considered in this Master thesis. This includes deciding upon a hardware topology which might include processing units and memory

units, pseudo components for avionics equipments such as sensors, actuators, storage memories and remote terminals and hardware interconnections such as buses, point-to-point links, serial lines etc. The execution platform services such as Monitoring and control (M&C) can be modeled as execution platform service instances and included in the hardware topology

- As only a small subset of possible non-functional properties are considered in this Master thesis, the realization of large number of possible non-functional properties are not considered. Non-functional properties such as worst-case execution time (WCET) bound for a certain operation, maximum memory footprint for a component implementation, communication budget allowed for an implementation, size of data types allowed in the communication etc., are yet to be effectively handled
- Even though the programming model in Chapter 5 discusses about handling the non-functional property `Bursty` which can be set on the provided interface side of the component offering the service, this is not part of the current version of the code generator due to time constraints in the Master thesis

Appendix A

A file structure for the generated code

Considering our running example for code generation in this Master thesis, the generated code is organized into the following files as explained below:

All the data types, event types, interfaces, exception types that are used in the example are stored along with the parameter channel and parameter queue as shown below:

```
src-gen
└── DatatypesInterfacesEventsAndExceptions
    └── include
        ├── Datatypes.h
        ├── Exceptions.h
        ├── FailureEvent.h
        ├── InterfaceA.h
        ├── InterfaceB.h
        ├── ParameterChannel.h
        └── ParameterQueue.h
```

All the constituents of the `Component_Callee` are arranged as shown below:

```
src-gen
├── Component_Callee
│   ├── AutogeneratedCode
│   │   ├── include
│   │   │   ├── ComponentType_Callee.h
│   │   │   └── EventEmitterPorts_Callee.h.h
│   │   └── src
│   │       ├── ComponentType_Callee.cpp
│   │       └── EventEmitterPorts_Callee.cpp
│   └── UserCode
│       ├── include
│       │   └── ComponentImplementation_Callee.h
│       └── src
│           └── ComponentImplementation_Callee.cpp
└── Component_Callee_impl_inst_Instance
    └── AutogeneratedCode
        ├── include
        │   ├── ProvidedInterfacePorts_Callee_impl_inst.h
        │   └── ComponentContainer_Callee_impl_inst.h
        └── src
            ├── ProvidedInterfacePorts_Callee_impl_inst.cpp
            └── ComponentContainer_Callee_impl_inst.cpp
```

All the constituents of the `Component_Caller` are arranged as shown below:

```
src-gen
├── Component_Caller
│   ├── AutogeneratedCode
│   │   ├── include
│   │   │   ├── ComponentType_Caller.h
│   │   │   ├── EventReceiverPorts_Caller.h
│   │   │   └── RequiredInterfacePorts_Caller.h
│   │   └── src
│   │       ├── ComponentType_Caller.cpp
│   │       ├── EventReceiverPorts_Caller.cpp
│   │       └── RequiredInterfacePorts_Caller.cpp
│   └── UserCode
│       ├── include
│       │   └── ComponentImplementation_Caller.h
│       └── src
│           └── ComponentImplementation_Caller.cpp
└── Component_Caller_impl_inst_Instance
    ├── AutogeneratedCode
    │   ├── include
    │   │   ├── ProvidedInterfacePorts_Caller_impl_inst.h
    │   │   └── ComponentContainer_Caller_impl_inst.h
    │   └── src
    │       ├── ProvidedInterfacePorts_Caller_impl_inst.cpp
    │       └── ComponentContainer_Caller_impl_inst.cpp
```

Appendix B

Additional OBSW examples

The code generator implemented as a part of this Master thesis, is also tested with other example OBSW models, designed using the OSRA SCM Model editor. Each of these examples are carefully constructed to capture all the corner cases listed in the section Section 6.3.1. The complexity of each of these models is much higher when compared to the running simple OSRA example model.

B.1. Producer/Consumer problem

This problem being one of the small collection of standard, well-known problems in concurrent programming domain. In this classical problem [**ProducerConsumer**], there are two entities specifically producers and consumers. Producers put items into the buffer and the consumers take items out of the buffer. Additionally a producer must wait until the buffer has space before it can put something in, and a consumer must wait until something is in buffer before it can take something out. In this example, we have two producers and one consumer. The figures in the Appendix B.1 and the tables Table B.1, Table B.2 and Table B.3 give an idea about how this example is constructed.

Table B.1.: Desired interaction kind for operations in the required interface ports

Component type	Required interface ports	Operations	Interaction kind
Producer_Synchronous	Consumer_IF_RI	ConsumeData	synchronous
Producer_Asynchronous	Consumer_IF_RI	ConsumeData	asynchronous

Table B.2.: Non-functional properties for the operations in the provided interface slots

Provided interface slot	Operation	Non-functional property
Producer_IF_PISlot1	StartProducingData	Cyclic, Period = 2s
Producer_IF_PISlot2	StartProducingData	Cyclic, Period = 3s
Consumer_IF_PISlot	ConsumeData	Protected

Table B.3.: Non-functional property for event reception

Component type	Event receiver slot	Event	Non-functional property
Producer_Synchronous	ConsumerFailure_RecSlot	ConsumerFailure	Protected
Producer_Asynchronous	ConsumerFailure_RecSlot	ConsumerFailure	Unprotected

B.2. Building block approach

The Component based approach is one of the high level requirements discussed in the Chapter 2. According to this requirement, it should be possible to design the software as a combination of reusable units. The reusable units, being component instances, the aim of this example to reiterate the CBSE approach by having multiple component instances which correspond to the same component type. The figures in the Appendix B.2 and the tables Table B.4 and Table B.5 give an idea about how this example is constructed.

Table B.4.: Desired interaction kind for operations in the required interface ports

Required interface ports	Operations	Interaction kind
AOCS_MODE_IF_RI1	Enable_Trans_To_Nom	asynchronous
	AOCS_State setter	synchronous
	AOCS_State getter	asynchronous
AOCS_MODE_IF_RI2	Enable_Trans_To_Nom	synchronous
	AOCS_State setter	asynchronous
	AOCS_State getter	synchronous

B.3. Component chaining

As discussed before in the initial chapters, the composability and compositionality are one the corner-stone principles of the OSRA. In line with these corner-stone principles, this example aims to chain different types of components together. The figures in the Appendix B.3 and the tables Table B.6, Table B.7 and Table B.8 give an idea about how this example is constructed.

Table B.5.: Non-functional properties for the operations in the provided interface slots

Provided interface slot	Operation	Non-functional property
Mode_Manager_IF_PISlot	StartStep	Cyclic, Period = 4s
AOCS_MODE_IF_PI1Slot	Enable_Trans_To_Nom	Sporadic, MIAT = 2s
	AOCS_State getter	Protected
	AOCS_State setter	Protected
AOCS_MODE_IF_PI2Slot	Enable_Trans_To_Nom	Protected
	AOCS_State getter	Protected
	AOCS_State setter	Protected

Table B.6.: Desired interaction kind for operations in the required interface ports

Required interface ports	Operations	Interaction kind
AOCS_MODE_IF_RI1	Set_Mode	asynchronous
	ExecuteTransitionToNom	synchronous
	previousMode getter	asynchronous
	previousMode setter	synchronous
	currMode getter	synchronous
	currMode setter	asynchronous
AOCS_MODE_IF_RI2	Set_Mode	asynchronous
	ExecuteTransitionToNom	asynchronous
	previousMode getter	synchronous
	previousMode setter	asynchronous
	currMode getter	asynchronous
	currMode setter	synchronous
POWER_IF_RI	powerValue getter	asynchronous
	powerValue setter	asynchronous
PLANNER_IF_RI	calculateNewPlan	synchronous

Table B.8.: Non-functional property for event reception

Event receiver slot	Event	Non-functional property
AOCSFailure_RecSlot	AOCSFailure	Unprotected

Table B.7.: Non-functional properties for the operations in the provided interface slots

Provided interface slot	Operation	Non-functional property
MM_CYCLE_IF_PISlot	MM_Step	Cyclic, Period = 2s
AOCS_MODE_IF_PI1Slot	Set_Mode	Sporadic, MIAT = 3s
	ExecuteTransitionToNom	Unprotected
	previousMode getter	Protected
	previousMode setter	Protected
	currMode getter	Protected
	currMode setter	Protected
AOCS_MODE_IF_PI2Slot	Set_Mode	Protected
	ExecuteTransitionToNom	Protected
	previousMode getter	Protected
	previousMode setter	Protected
	currMode getter	Protected
	currMode setter	Protected
AOCS_CYCLE_IFPIslot	Step	Cyclic, Period = 4s
PLANNER_IF_PISlot	CalculateNewPlan	Unprotected
POWER_IF_PISlot	powerValue getter	Protected
	powerValue setter	Protected

B.4. Cyclic dependency

This example takes into consideration the situation wherein the components are dependent on each other in such a way that there is a cyclic dependency. The expectation that this cyclic dependency should still be handled by the code generator efficiently is tested in this example. The figures in the Appendix B.4 and the tables Table B.9, Table B.10 and Table B.11 give an idea about how this example is constructed.

Table B.9.: Desired interaction kind for operations in the required interface ports

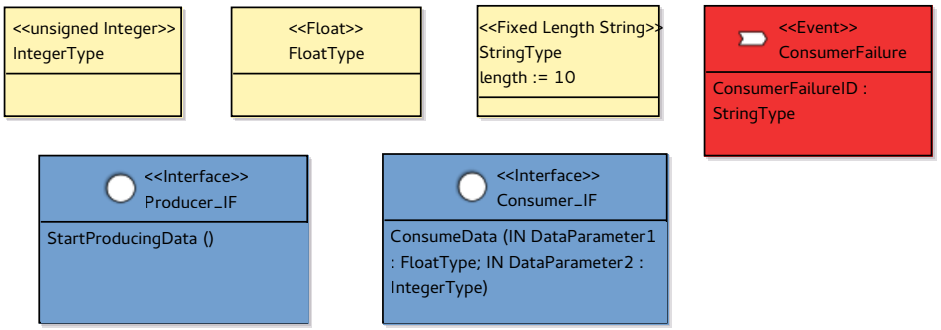
Component type	Required interface ports	Operations	Interaction kind
AOCS	PowerSubsystemInterface_RI	SwitchOnPower	asynchronous
		PowerSubsystemStatus getter	synchronous
		PowerSubsystemStatus setter	synchronous
AOCS	ThrusterSubsystem_RI	Send_THR_Pulse_CMD	synchronous
		ThrusterStatus getter	asynchronous
		ThrusterStatus setter	asynchronous
PowerSubsystem	DataHandlingAndMissionManagement_RI	HandleThisData	asynchronous
		StartMissionManagement	synchronous
ThrusterSubsystem	DataHandlingAndMissionManagement_RI	HandleThisData	synchronous
		StartMissionManagement	asynchronous

Table B.10.: Non-functional properties for the operations in the provided interface slots

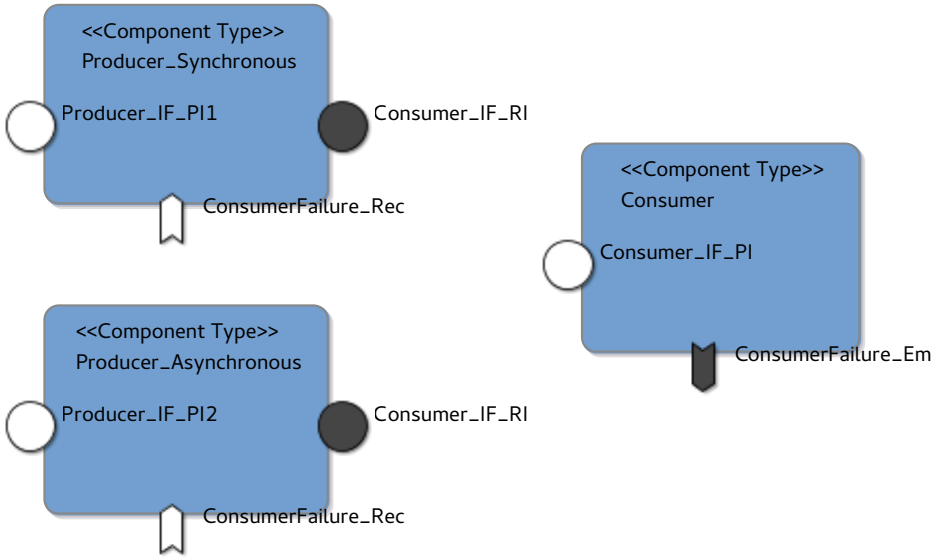
Provided interface slot	Operation	Non-functional property
AOCSInterface_PISlot	StartOperation	Cyclic, Period = 2s
DataHandlingAndMissionManagement_PISlot	HandleThisData	Protected
	StartMissionManagement	Protected
PowerSubsystemInterface_PISlot	SwitchOnPower	Sporadic, MIAT = 2s
	PowerSubsystemStatus getter	Protected
	PowerSubsystemStatus setter	Protected
ThrusterSubsystem_PISlot	Send_THR_Pulse_Cmd	Unprotected
	ThrusterStatus getter	Protected
	ThrusterStatus setter	Protected

Table B.11.: Non-functional property for event reception

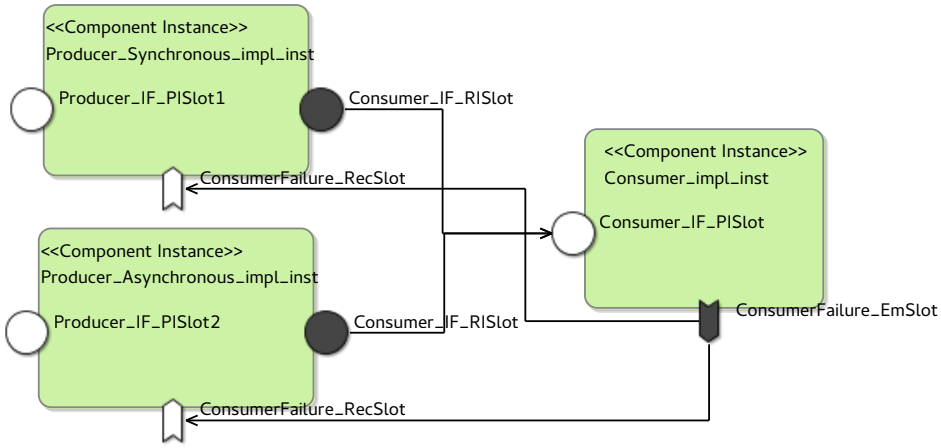
Event receiver slot	Event	Non-functional property
ThrusterSubsystemFailure_RecSlot	ThrusterSubsystemFailure	Protected
PowerSubsystemFailure_RecSlot	PowerSubsystemFailure	Protected



(a) Data types, events, exceptions and interfaces diagram

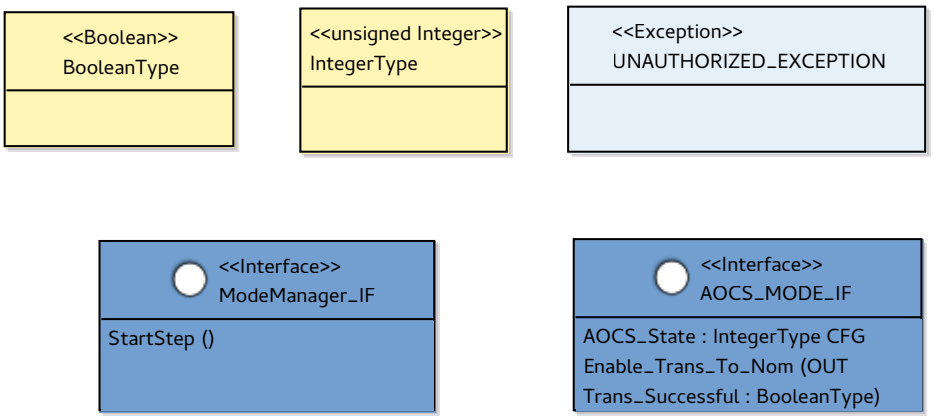


(b) Component types diagram

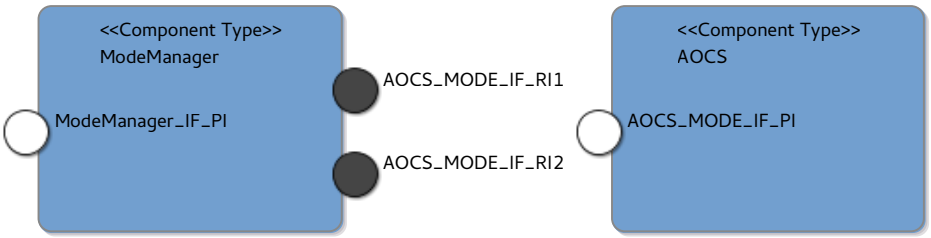


(c) Component instances diagram

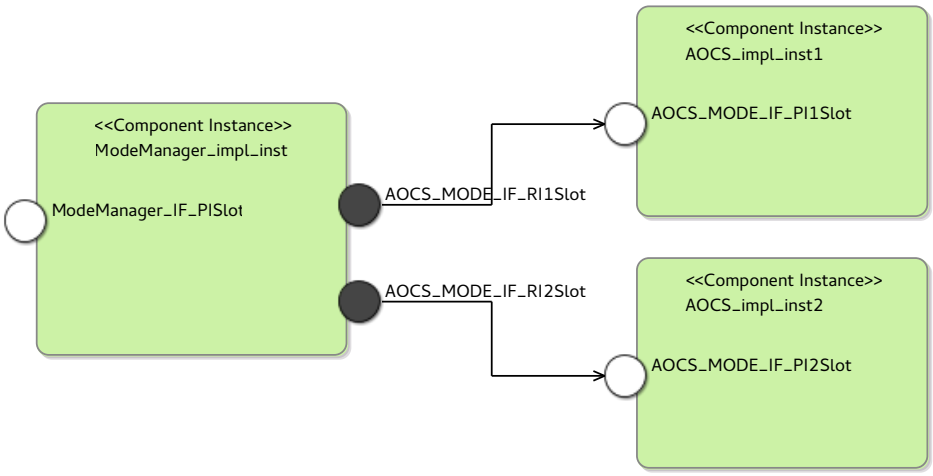
Figure B.1.: Producer/Consumer example



(a) Data types, events, exceptions and interfaces diagram

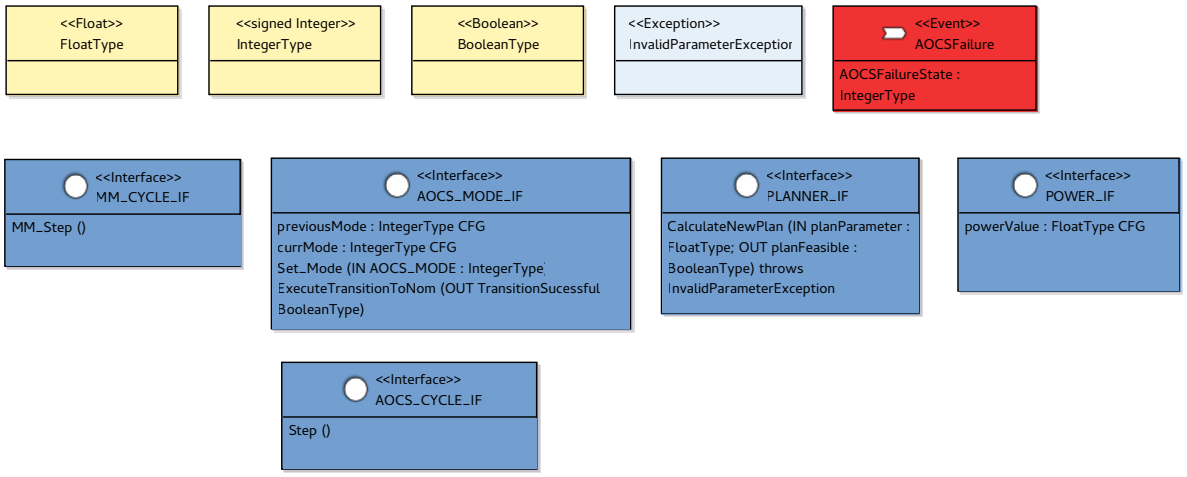


(b) Component types diagram

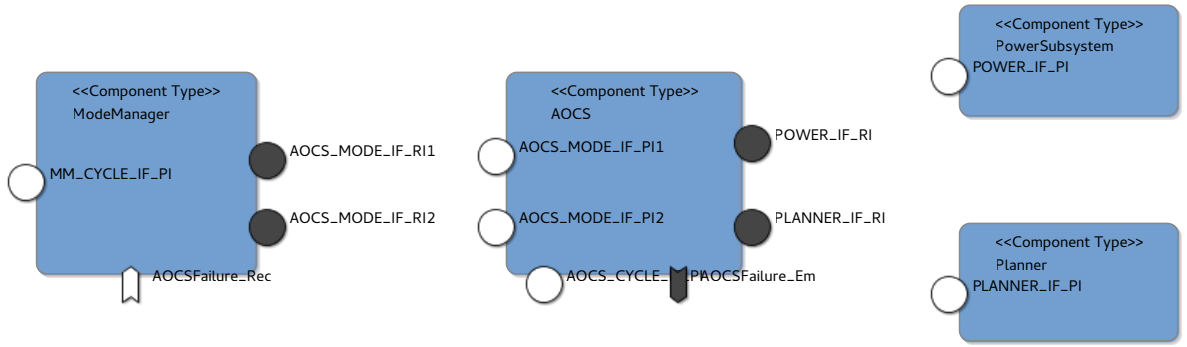


(c) Component instances diagram

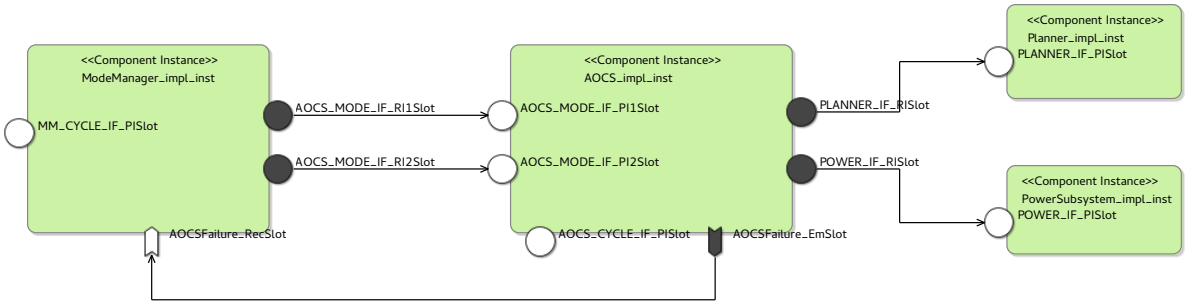
Figure B.2.: Building block approach example



(a) Data types, events, exceptions and interfaces diagram

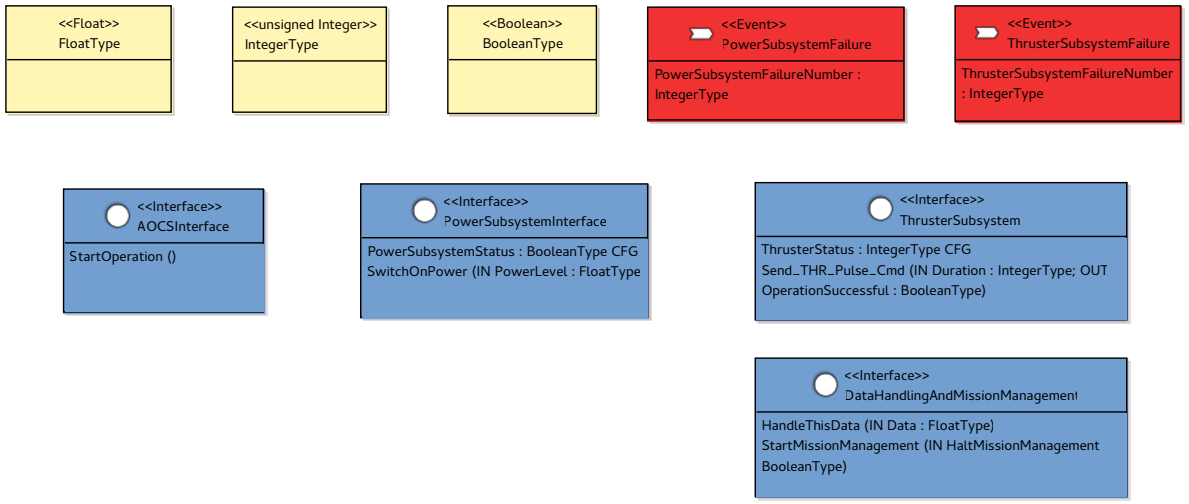


(b) Component types diagram

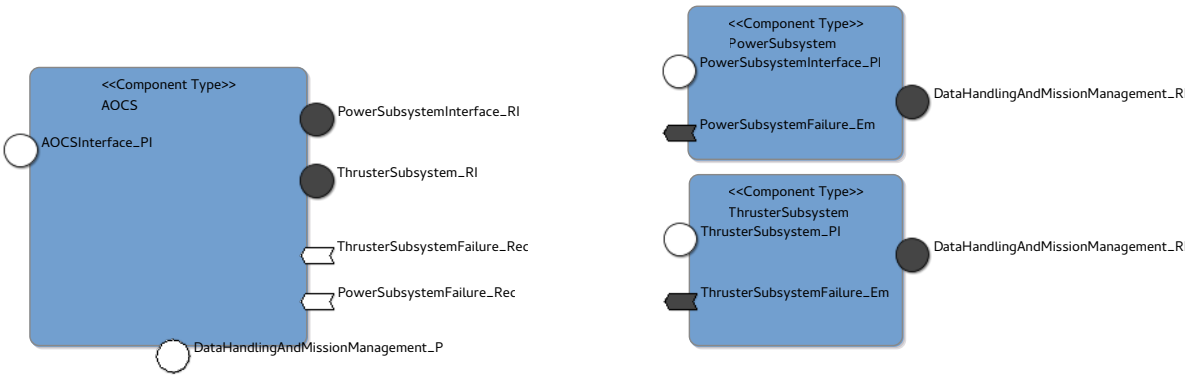


(c) Component instances diagram

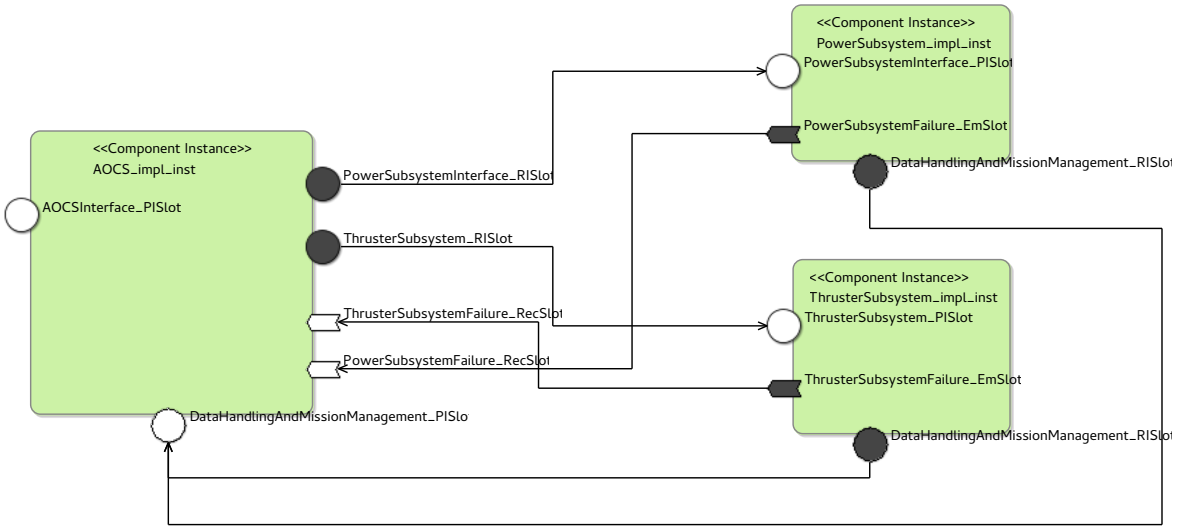
Figure B.3.: Component chaining example



(a) Data types, events, exceptions and interfaces diagram



(b) Component types diagram



(c) Component instances diagram

Figure B.4.: Cyclic dependency example

Appendix B

Bibliography

[1] M. P. Andreas Jung, J.-L. T. supported by the Savoir-Faire working group. *SAVOIR-FAIRE - On-board Software Reference Architecture*. Tech. rep. ESTEC-European Space Technology and Research Centre, 2010 (cit. on pp. 1, 3–8, 10, 12, 14, 18, 22, 29, 91).

[2] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2016. ISBN: 978-1-78646-496-5 (cit. on p. 86).

[3] M. Bordin, M. Panunzio, T. Vardanega. "Fitting Schedulability Analysis Theory into Model-Driven Engineering." In: *2008 Euromicro Conference on Real-Time Systems*. July 2008, pp. 135–144. DOI: 10.1109/ECRTS.2008.12 (cit. on pp. 10, 13, 18, 21, 42, 101).

[4] A. Burns, B. Dobbing, T. Vardanega. "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems." In: *Ada Lett.* XXIV.2 (June 2004), pp. 1–74. ISSN: 1094-3641. DOI: 10.1145/997119.997120. URL: http://doi.acm.org/10.1145/997119.997120 (cit. on p. 42).

[5] *C++11 styled callbacks?* URL: https://stackoverflow.com/questions/12338695/c11-styled-callbacks (cit. on p. 67).

[6] X. Cai, M. R. Lyu, K.-F. Wong, R. Ko. "Component-based software engineering: technologies, development frameworks, and quality assurance schemes." In: *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*. 2000, pp. 372–379. DOI: 10.1109/APSEC.2000.896722 (cit. on pp. 2, 11).

[7] S. Cammella-Dorda, J. Dean, G. Lewis, E. Morris, P. Oberndorf, E. Harper. *A Process for COTS Software Product Evaluation*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 2004 (cit. on p. 92).

[8] M. Carr, S. Konda, I. Monarch, C. F. Walker, C. Ulrich. *Taxonomy-Based Risk Identification*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 1993 (cit. on pp. 92, 93).

[9] *Code generation using Eclipse Xtend*. Cohesion force. URL: http://cohesionforce.github.io/reveal/#/ (cit. on p. 86).

[10] ESA, Obeo. *User Manual of the OSRA SCM Model Editor*. Tech. rep. European Space Agency, 2016 (cit. on pp. 32, 33, 60).

[11] P. Forgacs. *Custom C++ Exceptions for Beginners*. URL: http://peterforgacs.github.io/2017/06/25/Custom-C-Exceptions-For-Beginners (cit. on p. 63).

[12] M. Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. URL: https://martinfowler.com/articles/injection.html (cit. on pp. 16, 36, 61, 76, 86).

[13] G. Génova. *Modeling and Meta-modeling in Model Driven Development*. URL: http://www.ie.inf.uc3m.es/ggenova/Warsaw/Part3.pdf (cit. on p. 10).

[14] V. I. GmbH. *AUTOSAR Initiative*. URL: https://elearning.vector.com/index.php?&wbt_ls_seite_id=1044961&root=378422&seite=vl_autosar_introduction_en (cit. on pp. 2, 6).

[15] J. Grosse, S. Seidel, M. D. Lachmann, D. Becker, A. Wenzlawski, V. Schkolnik, A. N. Dinkelaker, O. Hellmig, H. Müntinga, M. Elsen, H. Ahlers, B. Weps, T. Wendrich, A. Stamminger, M. Krutzik, A. Peters, P. Windpassinger, E. Rasel, C. Braxmaier. "The MAIUS Sounding Rocket Missions – Recent Results, Lessons Learned and Future Activities." In: *68th International Astronautical Congress (IAC)*. Sept. 2017. URL: http://elib.dlr.de/115035/ (cit. on p. 38).

[16] W. D. H. Bo D. Hui, Z. Guifan. "Basic Concepts on AUTOSAR Development." In: *2010 International Conference on Intelligent Computation Technology and Automation*. (Changsha, China, May 11, 2010). IEEE, 2010, pp. 871–873. ISBN: 978-1-4244-7280-2. DOI: 10.1109/ICICTA.2010.571 (cit. on p. 3).

[17] *Inherit interfaces which share a method name*. URL: https://stackoverflow.com/questions/2004820/inherit-interfaces-which-share-a-method-name (cit. on p. 66).

[18] *Systems and Software engineering - Recomended practice for architectural description of software-intensive systems*. Standard. Geneva, CH: ISO/IEC 42010 (IEEE Std) 1471-2000, Mar. 2007 (cit. on p. 4).

[19] J. Klevin, H. Levinson, J. Marchetti. *Model-Driven Engineering: Automatic Code Generation and Beyond*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 2015 (cit. on pp. 91–93).

[20] P. Kruchten. *The Rational Unified Process: An Introduction – 2nd Edition*. Addison-Wesley Professional, 2000. ISBN: 0201707101 (cit. on p. 5).

[21] O. Maibaum, A. Heidecker. “Software Evolution from TET-1 to Eu:CROPIS.” In: *10th International Symposium on Small Satellites for Earth Observation*. Ed. by R. Sandau, H.-P. Röser, A. Valenzuela. Wissenschaft & Technik Verlag, Apr. 2015, pp. 195–198. URL: http://elib.dlr.de/100859/ (cit. on pp. 35, 36, 38).

[22] O. Maibaum, D. Lüdtke, A. Gerndt. “Tasking Framework: Parallelization of Computations in Onboard Control Systems.” In: *ITG/GI Fachgruppentreffen Betriebssysteme*. http://www.betriebssysteme.org/Aktivitaeten/Treffen/2013-Berlin/Programm/. Nov. 2013. URL: http://elib.dlr.de/87505/ (cit. on pp. 35–39, 100).

[23] O. Maibaum, T. Terzibaschian, C. Raschke, A. Gerndt. “Software Reuse of the BIRD ACS for the TET Satellite Bus.” In: *8th IAA Symposium on Small Satellites for Earth Observation*. Ed. by R. Sandau, H.-P. Röser, A. Valenzuela. Wissenschaft und Technik Verlag Berlin, Apr. 2011, pp. 409–412. URL: http://elib.dlr.de/68293/ (cit. on pp. 35, 36).

[24] S. McConnell. *Code Complete – 2nd Edition*. Microsoft Press, 2004. ISBN: 0-7356-1967-0 (cit. on p. 61).

[25] E. Mezzetti, M. Panunzio, T. Vardanega. “Temporal Isolation with the Ravenscar Profile and Ada 2005.” In: *Ada Lett.* 30.1 (May 2010), pp. 45–55. ISSN: 1094-3641. DOI: 10.1145/1806546.1806551. URL: http://doi.acm.org/10.1145/1806546.1806551 (cit. on pp. 100, 101).

[26] S. Montenegro, J. Richardson. “RODOS operating system for Network Centric Core Avionics.” In: (Feb. 2018) (cit. on p. 38).

[27] *Obeo Designer Framework*. URL: https://www.obeodesigner.com (cit. on p. 30).

[28] *Other Data Types*. URL: http://www.cplusplus.com/doc/oldtutorial/other_data_types/ (cit. on p. 63).

[29] M. Panunzio, T. Vardanega. “A Component Model for On-board Software Applications.” In: *36th EUROMICRO Conference on Software Engineering and Advanced Applications*. (Sept. 1, 2010). Lille, France: IEEE, pp. 57–64. ISBN: 978-1-4244-7901-6. DOI: 10.1109/SEAA.2010.39 (cit. on pp. 10, 14).

[30] M. Panunzio, T. Vardanega. “On Component-Based Development and High-Integrity Real-Time Systems.” In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2009, pp. 79–84. DOI: 10.1109/RTCSA.2009.15 (cit. on pp. 2, 9, 10, 12, 16, 28).

[31] M. Panunzio. “Definition, realization and evaluation of a software reference architecture for use in space applications.” PhD thesis. Universitàdi Bologna Università degli Studi di Padova, 2011 (cit. on pp. 1, 2, 4–8, 11–13, 15, 16, 19, 22, 29, 35, 42).

[32] M. Panunzio. *Specification of the Metamodel for the OSRA Component Model*. Tech. rep. Thales Alenia Space - France, 2017 (cit. on pp. 17, 31, 46, 49–51, 53–57, 59, 63, 65, 71, 76, 77, 85).

[33] M. Panunzio, T. Vardanega. "A component-based process with separation of concerns for the development of embedded real-time software systems." In: *Journal of Systems and Software* 96 (2014), pp. 105–121. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.05.076. URL: http://www.sciencedirect.com/science/article/pii/S0164121214001381 (cit. on pp. 10, 12–14, 16, 21–25, 27–31, 41, 42, 46, 49, 50, 55, 59, 70, 71, 76, 77, 79, 91, 101).

[34] M. Panunzio, T. Vardanega. "Ada Ravenscar Code Archetypes for Component-Based Development." In: *Reliable Software Technologies – Ada-Europe 2012*. Ed. by M. Brorsson, L. M. Pinho. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–17. ISBN: 978-3-642-30598-6 (cit. on pp. 41–43, 45, 50, 53, 62–64, 70, 71, 75, 76, 83, 101).

[35] M. Panunzio, T. Vardanega. "Charting the Evolution of the Ada Ravenscar Code Archetypes." In: *Ada Lett.* 33.1 (June 2013), pp. 64–83. ISSN: 1094-3641. DOI: 10.1145/2492312.2492320. URL: http://doi.acm.org/10.1145/2492312.2492320 (cit. on pp. 10, 13, 42, 45, 46, 50, 73).

[36] M. Panunzio, T. Vardanega. "On Software Reference Architectures and Their Application to the Space Domain." In: *Safe and Secure Software Reuse*. Ed. by J. Favaro, M. Morisio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–159. ISBN: 978-3-642-38977-1 (cit. on pp. 5, 10).

[37] T. Peng, K. Höflinger, B. Weps, O. Maibaum, K. Schwenk, D. Lüdtke, A. Gerndt. "A Component-Based Middleware for a Reliable Distributed and Reconfigurable Spacecraft Onboard Computer." In: *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2016, pp. 337–342 (cit. on p. 38).

[38] *Petri net*. URL: https://en.wikipedia.org/wiki/Petri_net (cit. on p. 37).

[39] A.-I. Rodriquez, F. Ferrero, E. Alana, A. Jung, M. Panunzio, T. Vardanega, A. Grenham. "The Component Layer of Cordet On-Board Software Architecture." In: *DASIA 2012 Data Systems In Aerospace*. (in Drubrovnik, Croatia, May 14, 2012). Ed. by L. O. E. SP-701. 2012. ISBN: 978-92-9092-265-0 (cit. on pp. 4, 7, 10).

[40] J. A. S. Dersten, J. Froberg. "Effect Analysis of the Introduction of AUTOSAR - A Systematic Literature Review." In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. (Oulu, Finland, Aug. 30, 2011). IEEE, 2011, pp. 239–246. ISBN: 978-1-4577-1027-8. DOI: 10.1109/SEAA.2011.44 (cit. on p. 3).

[41] T. Stahl, M. Völter, J. Bettin, A. Hasse, S. Helsen. *Model-Driven Software Development*. John Wiley & Sons, Ltd, 2006. ISBN: 978-0-470-02570-3 (cit. on p. 63).

[42] B. Stroustrup. *The Design and Evolution of C++*. Addision-Wesley Publishing Company, 1994. ISBN: 978-0201543308 (cit. on p. 66).

[43] S. Theil, N. A. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, J. Wohlfeil. "ATON - Autonomous Terrain-based Optical Navigation for Exploration Missions: Recent Flight Test Results." In: *Deutscher Luft- und Raumfahrtkongress*. Sept. 2017. URL: http://elib.dlr.de/114457/ (cit. on pp. 35, 38).

[44] K. Thoms. *Recipes to build Code Generators for Non-Xtext Models with Xtend*. URL: https://www.eclipsecon.org/europe2016/session/recipes-build-code-generators-non-xtext-models-xtend (cit. on p. 86).

[45] O. Tuna. *Multiple Inheritance and the Diamond Problem*. URL: https://medium.freecodecamp.org/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbbec (cit. on p. 76).

[46] *What Is Google C++ Mocking Framework?* URL: https://github.com/google/googletest/blob/master/googlemock/docs/ForDummies.md (cit. on p. 61).

[47] *Xtend - Documentation*. Xtend. URL: http://www.eclipse.org/xtend/documentation/ (cit. on p. 86).

All links were last followed on March 28, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature