Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Space Component Model using DLR Software Technologies

Raghuraj Tarikere Phaniraja Setty

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn (Prof.-Vertr.)
Supervisor:	Dr. Dušan Okanović, Teerat Pitakrat, M.Sc.
Commenced:	October 2, 2017
Completed:	April 2, 2018
CR-Classification:	I.7.2

Abstract

... Short summary of the thesis in English ...

Kurzfassung

... Short summary of the thesis in German ...

Contents

List of Figures

List of Tables

List of Acronyms

FR Fehlerrate

List of Listings

List of Algorithms

Chapter 1

Introduction

In diesem Kapitels steht die Einleitung zu dieser Arbeit. Sie soll nur als Beispiel dienen und hat nichts mit dem Buch [**WSPA**] [**SAVOIR**] zu tun. Nun viel Erfolg bei der Arbeit!

Bei LaTeX werden Absätze durch freie Zeilen angegeben. Da die Arbeit über ein Versionskontrollsystem versioniert wird, ist es sinnvoll, pro *Satz* eine neue Zeile im `.tex`-Dokument anzufangen. So kann einfacher ein Vergleich von Versionsständen vorgenommen werden.

Thesis Structure

Die Arbeit ist in folgender Weise gegliedert:

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

Kapitel 7 – Conclusion 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 [**SAVOIR**]. The implementation of such a vision is expected to provide benefits for all the stake-holders in the space community [**SAVOIR**].

Customer Agencies Significant reduction in the project development cost and schedule and the risk involved in the 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. AUTOSAR (AUTomotive Open System ARchitecture) for the automotive industry is worthy mentioning [**BasConAUTOSAR**]. 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 the sector and is driven by the need of the industry to become more competitive [**EfAnAUTOSAR**]

Space primes and on-board software companies have made significant progress and have implemented and/or are implementing 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 company level [**SAVOIR**]

ESA through its two parallel activities, namely COrDeT and DOMENG [**CORDET**] 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 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" Advisory Group (SAVOIR Advisory Group) was created. SAVIOR 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. OSRA is the result of the R&D activities of this group [**SAVOIR**].

The On-board software reference architecture (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 [**SAVOIR**]. 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.

2.2. Need for software reference architecture

2.2.1. Motivation

According to the ISO/IEC standard ISO 42010 [**ISO42010**], 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.

According to the "Rational Unified Process"(RUP)[**RUP**], 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 was developed. So, reference architecture is a form of "generic" software architecture which prescribes the founding principles, the underlying methodology and the architectural practices that were recognized by the domain stakeholders as the best solution to the construction of a certain class of software systems [**PhdThesis**; **SoftRefArch**].

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 [**SAVOIR**].

A software reference architecture is made up of two main parts: [**SAVOIR**]

- A software architectural concept addressing the pure software architectural related issues
- Architectural building blocks 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: [**SAVOIR**]

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 functionality 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 [**SAVOIR**]. 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:

- 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, read 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 [**PhdThesis**]. 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 the suitable complete implementation. Development of interface specifications with reference architectures as the basis allows the implementation of the famous AUTOSAR concept: "*Cooperate on standards, compete on implementation*"[**AUTOSARurl**]

The OBSW life-cycle needs to be consistent with the system life-cycle, which features the definition of functional increments in system development [**SAVOIR**]. 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 [**CORDET**; **SAVOIR**]. These stakes are included and defined as the user needs [**SAVOIR**; **PhdThesis**] for the development of OSRA:

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

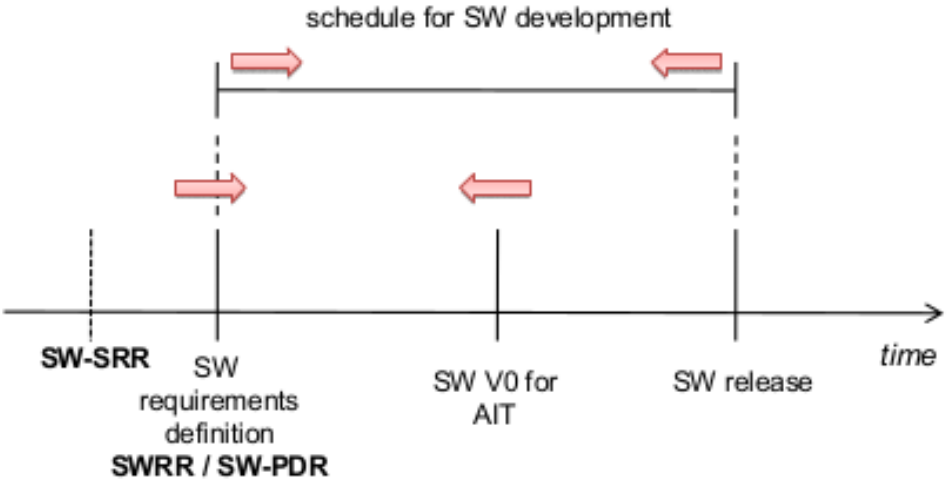


Figure 2.2.: Reduction in the schedule for software development

Source: [**SAVOIR**]

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 with a certain amount of budge.

Reduce Verification and Validation effort The new development approaches shall foster the reduction of effort for Verification and Validation efforts, 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 amongst 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 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 [**SAVOIR**; **PhdThesis**].

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 [**CompBasedDev**]. Chapter 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 [**CompBasedDev**]. The validation-intensive real-time high-integrity systems such as on-board software systems make the adoption of the MDE considerable 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 'CHESS: space case study' [**CompBasedProcess**] and the 'ESA: reference Earth Observation case study' [**CompBasedProcess**].

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 [**Metamodelling**].

COrDeT (Component Oriented Development Techniques) aimed at investigating various techniques in fields such as software product line engineering, model driven engineering and component orientation. The study came up with the concept of software reference architecture which is to be made up of: [**CORDET**; **SoftRefArch**; **SAVOIR**]

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

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 [**ScheduAnaly**]

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.[**CharEvoRAVCodeAr**]

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.

2.3.2. Component Model

Component Based Software Engineering (CBSE) is a software methodology centered on the systematic re-use of software by realizing the software as assembly of nits of composition called components [**CBSE**]. 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 [**SAVOIR**] because of strong verification and validation requirements imposed in the space domain and the presence of non-functional dimensions [**SAVOIR**].

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 [**SAVOIR**]:

- 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 do not appear in the design space)
- Possibility of verification of properties related to composability and compositionality [**CompBasedDev**]

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).

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 [**CompBasedProcess**]. 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 leads to safer, cheaper and more reliable software [**CompBasedProcess**; **PhdThesis**]. 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) [**CompBasedProcess**] wherein:

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

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

- 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 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 [**ScheduAnaly**].

- 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 [**CharEvoRAVCodeAr**].

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

- It increases the reuse potential of the components, which is an important high level requirement described in the previous section [**SAVOIR**]. 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 on the run-time behavior of the software. 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 [**CompBasedProcess**]. 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, deployment on target and execution. The components, as mentioned before, implement 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 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 [**SAVOIR**; **ComponentModel**]. 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.

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 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.

In OSRA, the first and second needs can be met by having correct representation of 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.

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 [**CompBasedProcess**].

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 pf development [**PhdThesis**].

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" [**CompBasedProcess**; **InvOfCntrlurl**]. 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 interface [**CompBasedDev**]. 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" [**CompBasedDev**]. 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.

2.3.3. Computational model

Using a computational model is necessary as per the Space Software engineering standard (ECSS-E-ST-440C) standard [**SAVOIR**]. 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 [**ScheduAnaly**]. 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 [**SAVOIR**]:

- 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/kernel (RTOS/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 [**SAVOIR**]:

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 Section 2.3.2)

Services to implement "abstract components" These services include 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 [**PhdThesis**]:

- 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 [**PhdThesis**].

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 [**CompBasedProcess**]. 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 [**ScheduAnaly**]. 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 [**SAVOIR**; **CompBasedProcess**]. As already mentioned in the previous chapter, this master thesis considers Tasking Framework as the computational model and a normal linux based machine 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 [**CompBasedProcess**; **PhdThesis**; **SAVOIR**]:

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`,`setter` operations for attributes with `read-write` access.

Step 3: Definition of component types Component types form the basis of a reusable software asset [**CompBasedProcess**]. The software architect defines the component type to provide the specification of the functions that the component of this 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 this type would provide
- One or more required interfaces, which list the functional services that the component of this type would require in order to function correctly according to the functional specifications

- A set of component type attributes of already defined data types that are local to the component 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

Step 4: Definition of component implementations The software architect now creates and refines 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. 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 [**CompBasedProcess**].

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 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 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 RI (required interface) and the called operation in the PI (provided interface) 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.

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

In this step, the software architect specifies the timing and the synchronization attributes [**CompBasedProcess**]. 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.

An `immediate` 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 `deferred` 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 the execution platform and the infrastructure code guarantees this and the component implementer does not have to worry about it, as in the case for sporadic operation.

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.

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

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, the 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 (refer Step 8). In majority of the cases, it is straight-forward to allocate the bindings between the components as they are deployed on the same processing units [**CompBasedProcess**]. 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 PIM (Platform Independent Model), a Schedulability Analysis Model (SAM) which is a PSM (Platform Specific Model) 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 [**CompBasedProcess**].

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. 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.

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 and the component instance and component connector 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.

3.3. Design flow and design views

When the component model is defined, it also defines implicitly a design flow, that needs to be followed, to be able to create an OBSW that meet all its user needs and high level requirements [**SAVOIR**; **PhdThesis**; **CompBasedProcess**]. The design flow is as explained in the previous section.

The Obeo Designer Framework provides a concept called 'Viewpoint' and using this concept, the design views are implemented [**CompBasedProcess**]. One of the advantages of the design views is to promote or enforce a certain design flow [**CompBasedProcess**]. 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 [**SpecMetamodel**]. Each language unit consists of closely related metamodel entities. OSRA Component model is composed of the following language units:

`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 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.

The toolset is available as

- 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 [**OSRAEditor**].

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 instance diagram

Component Instance 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. Tables are usually 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 the reference [**OSRAEditor**].

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 [**PhdThesis**]. 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 for extra terrestrial navigation and landing systems [**ATON**]. 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 [**TaskFr**]. Because of this and the uncertainty of propulsion and parachute maneuver, the amount of different landing targets with low risks are considerably reduced [**TaskFr**]. 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 is is also needed to integrate hazard avoiding algorithms [**ATON**]. However, these algorithms have a huge demand for computing power [**TaskFr**].

In TET-1 satellite mission (Technology demonstrator) and BIRD (Bi-spectral Infrared Detection) missions, the estimator and predictor modules were computed in a fixed order and fixed time in the control-cycle [**TETBIRD**]. 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 [**TETBIRD**; **TETtoEUCROPIS**]. During the launch and early orbit phase (LEOP), a timing violation in another bus application occurred which resulted in an unexpected AOCS state caused by change in the ordering of inter-dependent computations and corrupted data [**TETBIRD**].

On the other hand, the current on-board systems for the space environment do not provide the needed computing power [**TaskFr**]. 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 [**TaskFr**]. Hence a concept, which allows sharing of computing resources based on predefined configurations for different flight phases and fault scenarios is necessary [**TaskFr**]. The Tasking framework is an incarnation of the Inversion of Control design pattern which is popular practice in lightweight container frameworks [**InvOfCntrlurl**].

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 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 petrinets, where the activation of the task input is a token and the task execution is a transition [**TaskFr**]. 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 [**TaskFr**]. 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. Task event can be configured to work with absolute timing, i.e. the task input and in-turn the task is activated at fixed points in time, or the 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.

The current implementation of Tasking framework sits on top of Linux POSIX library, composed by a real-time clock interface, signaling mechanism, memory access and the tasks scheduler. The Tasking framework can also run on RTEMS and FreeRTOS using the outpost libraries which collectively provide a high level abstraction of the underlying operating system [**TaskFr**].

4.3. Use cases for Tasking Framework

In the project OBC-NG (On-Board Computer - Next Generation) by DLR, a decision is made to design the on-board computer systems as a combination of space qualified processing node, COTS (Commercial off-the-shelf) processing nodes and network nodes [**TaskFr**]. As an operating system for this project, an enhancement of RODOS is used [**TaskFr**; **OBC-NG**]. The enhancement covers mainly the support for multi-core and re-configurable distributed systems [**RODOS**] and the core element which makes it possible is the Tasking Framework [**TaskFr**]. The configuration manager used in OBC-NG holds predefined mappings of tasks and resources for different hardware configurations of computing resources and mission phases [**OBC-NG**]. The communication infrastructure would be set up based on the mappings during the configuration phases of the system.

The first usage of the Tasking framework was in the ATON (Autonomous Terrain-based Optical navigation) project [**ATON**]. 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 [**ATON**].

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

Tasking framework is also used in MAIUS mission (Matterwave Interferometer in Microgravity) which deals with activities to demonstrate Bose-Einstein condensation

and atom interferometry with rubidium and potassium atoms on a sounding rocket [**TETtoEUCROPIS**; **MAIUS**].

4.4. Use of Tasking Framework in this thesis

Tasking framework is used as a computational model, as discussed in the previous chapter. The reasons for adopting 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 (refer previous section on use cases for Tasking framework)
- 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 FIFO scheduling for tasks with same priorities which uses these shared resources [**TaskFr**].

However, during the course of this Master thesis, certain short-comings of Tasking framework have been identified and they would be topics for discussion in the upcoming chapters

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 component model proposed [**CompBasedProcess**; **EvoRAVCodeAr**]. 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 the Step 7 (Specification of non-functional attributes) 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, 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 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 ASSERT project (Automated proof-based System and Software Engineering for Real-Time systems), was the first, large project which showed the feasibility of a development approach for high-integrity real-time systems centered on separation of concerns, correctness-by-construction and property preservation.

In the ASSERT project, which incorporated Model-driven-engineering approaches [**PhdThesis**], a modelling infrastructure named RCM was developed at the University of Padua [**ScheduAnaly**]. This infrastructure included a graphical modeling language and an editor, a model validator and set of model transformations that were necessary to feed model-based analysis and code generation [**ScheduAnaly**]. The Ravenscar Computational Model (RP) was chosen as a computational model in this modelling infrastructure and RP directly emanated from the Ada Ravenscar Profile in language-neutral terms [**EvoRAVCodeAr**]. The RP basically does not allow any language constructs that are exposed to unbounded execution-time and non-determinism [**CharEvoRAVCodeAr**; **EvoRAVCodeAr**; **RAVCodeAr**]. Certain RP-compliant code archetypes were developed to complete the formulation of a programming model in the ASSERT project, which adhered to the vision of principle of separation of concerns and amenable to code generation [**CharEvoRAVCodeAr**]. The code archetypes used Ada run-time and hence could fit the needs of typical embedded systems which were resource-constrained [**RAVCodeAr**]. The archetypes developed in the ASSERT project, were based on the previous work on code generation from HRT-HOOD to Ada [**CharEvoRAVCodeAr**; **EvoRAVCodeAr**].

In the following Artemis JU CHESS project, which was an initiative from ESA in parallel to the development of the SCM [**PhdThesis**; **CompBasedProcess**], these code archetypes from the ASSERT were revised by adding certain features [**EvoRAVCodeAr**]. The code archetypes in the CHESS project also targeted the Ravenscar Computational Model 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 [**CharEvoRAVCodeAr**]. The code archetypes developed in the CHESS project [**EvoRAVCodeAr**] 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 at the end 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 strive to attain as much separation as possible between the functional and extra-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.

The 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. The mapping of this structure to the actual design entity of the infrastructural code is not of concern in 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 environement, and the provided services invoked by the external clients are executed with the desired interaction semantics.

As multiple clients may independently require the 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. Safeguarding of execution requests in case of synchronous service requests is implicit as these requests would have been raised in the respective flow of control of the 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 data 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. 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.

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.

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 component handling the request. Along with the data associated with the service request, it is also important that the required interface port packages information about how to send back the results and exceptions (if any) to the service requester.

Each thread of control, having its own structure as explained below, is responsible for only one operation in the provided interface side of the component that handles requests. 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 archetype listed in [**CharEvoRAVCodeAr**; **EvoRAVCodeAr**] 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 [**CharEvoRAVCodeAr**].

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) [**SpecMetamodel**; **CompBasedProcess**]. 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. 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.

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 requests, is a task with one task input attached to the task. 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.

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 [**SpecMetamodel**; **CompBasedProcess**].

The general structure of the thread of control on the service provider end, necessary to handle this kind of service requests is a task with two non-synchronized task inputs attached to it. 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 configured in a way 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 [**CharEvoRAVCodeAr**; **EvoRAVCodeAr**]. It is unclear from the available resources whether, the reason to not consider this code archetype, was because of the non-possibility to opt this non-functional property for a service request on the provided interface side.

Cyclic

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

The general structure of the thread of control on the service provider end, necessary to handle this kind of service requests is a task with a task event provided by the Tasking framework, attached to it. The task event is configured to wake up the 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 [**SpecMetamodel**]. The services also need to be parameterless and cannot send out results or throw exceptions [**SpecMetamodel**].

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

Chapter 6

Infrastructural code generation

6.1. Introduction

After designing an OBSW model using the OSRA editor and following the component-based software development approach that comes with it, the OBSW model entities need to be mapped to the 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 briefly 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 [**EvoRAVCodeAr**], 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 [**SpecMetamodel**]. However a brief description of them is useful here:

Datatypes The software architect can create a set of project-specific data types and constants using the Datatypes language unit of the CommponTypes 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).[**SpecMetamodel**]. 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 the data type definitions are obvious for readers with programming skills in typed languages such as Ada, C or C++.

Interfaces An interface is a specification of 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, 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 Component type is an entity which specifies the external interfaces of a software component and 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. Component type usually encompasses:

- List of provided interface ports
- List of required interface ports
- List of dataset emitter ports
- List of dataset receiver ports
- List of event emitter ports
- List 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 (Different 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 [**SpecMetamodel**].

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 mapping of design entities to the infrastructure code that will be generated in C++. Certain terms specific to C++ only, are used in this section.

On analyzing the specification of the metamodel for the OSRA component model [**SpecMetamodel**], it is clear that there might be different corner cases that might possibly 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 might arise are:

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

The first and second corner cases are handled in the following example. But, the other cases 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 Chapter 3. As already mentioned in that section, the component-based approach puts a lot of emphasis on the definition of component interfaces

[**CompBasedProcess**] 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 [**SpecMetamodel**]. The OBSW model is designed using the OSRA model editor mentioned in the section Section 3.5 in chapter Chapter 3.

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.

Two datatypes namely `FixedLengthStringType` and `IntegerType` (with `integerKind` set to `UNSIGNED`) are defined and they are named as `StringType` and `IntegerType` respectively. Three exception types, named as `OperandException`, `MemoryException` and `OverflowException` are defined. An `Event` type, which can be used for asynchronous notifications [**SpecMetamodel**] is instantiated and it is named as `FailureEvent`. Two parameters namely `m_Param` and `m_Description` with data types `IntegerType` and `StringType` respectively are instantiated as parameters of the `FailureEvent`

Step 2: Definition of interfaces Two interface namely `InterfaceA` and `InterfaceB` are designed. `InterfaceA` has only one single operation by name `CallOperationAdd` and `InterfaceB` has an operation by name `OperationAdd` and an interface attribute of data type `IntegerType` and named as `m_StatusValue`.

The `OperationAdd` has three parameters, out of which two parameters have `ParameterDirectionKind` set to `in` and the third parameter has the `Parameter DirectionKind` set to `out`. The `OperationAdd` is also configured to return any of the three exception kinds mentioned in the previous step. The interface attribute `m_StatusValue` has the `AttributeKind` set to `CFG` which indicates that the interface attribute parameter is a configurable parameter `SpecMetamodel`. As a result, two operations for the purpose of setting and getting the values of the interface attribute are defined.

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.

`Component_Caller` has one provided interface port named as `ProvidedInterface Port` and two required interface ports named as `RequiredInterfacePortType1` and `RequiredInterfacePPortType2`. `ProvidedInterfacePort` refers to `InterfaceA`

and both RequiredInterfacePortType1 and RequiredInterfacePortType2 refer to InterfaceB. All the operations in the RequiredInterfacePortType1 have the desired interaction kind set to synchronous and all the operations in the RequiredInterfacePortType2 have the desired interaction kind set to asynchronous (Note that, it is also possible to independently choose the desired interaction kind for each operation [**SpecMetamodel**]). Component_Calller also has one event receiver port, named as FailureEventReceiverPort and it refers to the FailureEvent.

Component_Callee has two provided interface ports named as ProvidedInterface Port1 and ProvidedInterfacePort2 and no required interface port. Both ProvidedInterfacePort1 and ProvidedInterfacePort2 refer to InterfaceB. Component_Callee also has an event emitter port called the FailureEvent EmitterPort which refers to the FailureEvent.

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 m_Param of InterfaceB, that is exposed through its provided interface ports namely ProvidedInterface Port1 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.

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

Two component instances namely Component_Caller_impl_inst and Component _Callee_impl_inst are instantiated from the component implementations Component_Caller_impl and Component_Callee_impl respectively.

Step 6: Definition of component bindings The following component bindings are defined:

- The required interface slot RequiredInterfaceSlotType1 of the component instance Component_caller_impl_inst is connected to the provided interface slot ProvidedInterfaceSlot1 of the component instance Component _callee_impl_inst

- The required interface slot `RequiredInterfaceSlotType2` of the component instance `Component_caller_impl_inst` is connected to the provided interface slot `ProvidedInterfaceSlot2` of the component instance `Component_callee_impl_inst`
- The event emitter slot `FailureEventEmitterSlot` of the component instance `Component_callee_impl_inst` is connected to the required interface slot `FailureEventReceiverSlot` of the component instance `Component_caller_impl_inst`

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 [**SpecMetamodel**]. VSL is used here to define values of non-functional properties with a measurement unit.

The operations in the provided interface slots, that are instances of the provided interface ports, that are promoted by the component containers of the respective components are assigned different non-functional properties namely:

- The operation `CallOperationAdd` provided by the `ProvidedInterfaceSlot` which refers the `InterfaceA`, in component instance `Component_caller_impl_inst` is marked as a cyclic operation with period as 2*s*
- The operations in the provided interface port `ProvidedInterfaceSlot1` which refers to the interface `InterfaceB`, in component instance `Component_callee_impl_inst`, are marked with the following non-functional properties:
  - The operation `OperationAdd` is marked as a protected operation
  - The interface attribute setter operation for the interface attribute `m_StatusValue` is marked as a protected operation
  - The interface attribute getter operation for the interface attribute `m_StatusValue` is marked as an unprotected operation
- The operations in the provided interface port `ProvidedInterfaceSlot2` which refers to the interface `InterfaceB`, in component instance `Component_callee_impl_inst`, are marked with the following non-functional properties:
  - The operation `OperationAdd` is marked as a sporadic operation with MIAT as 2*s*

- The interface attribute setter operation for the interface attribute `m_StatusValue` is marked as a protected operation
- The interface attribute getter operation for the interface attribute `m_StatusValue` is marked as a protected operation

The event receiver slot `FailureEventReceiverSlot`, which is an instantiation of the event receiver port `FailureEventReceiverPort`, in the component instance `Component_caller_impl_inst`, is set with the reception of the event `FailureEvent` as a protected operation.

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.

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

A processor board with a processor and a processor core is designed. 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 [**OSRAEditor**]. Only after the OBSW model is successfully validated, can the OBSW model be considered as a suitable candidate for automatic generation of infrastructure code [**OSRAEditor**].

6.3.3. Software design approach for the generated code

This section deals with the software design approach for the generated infrastructure code. UML class diagrams are used to show a high level representation of the generated C++ classes.

Data types and events

A data type from the OBSW model is translated into simple `typedef` statement from C++.

In case of the above example:

- The data type `IntegerType`, is translated to `typedefint8_t IntegerType`

- The data type `StringType`, is translated to `typedefstd::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++. These exceptions, which can be thrown by a particular operation are grouped under an enumeration.

In case of the above example, the three exceptions `OperandException`, `MemoryException` and `OverflowException` and each exception is translated into an enumeration literal which has the same name as the corresponding exception. Three 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 `OperationAddInterfaceBException`.

An event from the OBSW model is mapped to an abstract base class and a concrete implementation class. 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 the concrete implementation corresponding to the evnt.

In case of the above example, the `FailureEvent` is mapped as an abstract base class named `FailureEventInterface` and concrete implementation class named `FailureEvent`. Appropriate setters and getters for the event parameters `m_Param` and `m_ParamDescription` are declared as pure virtual methods in the `FailureEvent Interface` abstract base class and redefined in the `FailureEvent` concrete implementation class.

All the infrastructure code entities mentioned above are present in a namespace, named as `General`.

Interfaces

An interface can be mapped to an abstract base class in C++. Constituents of this abstract base class are:

- For each interface operation, corresponding operation parameters and corresponding data types of the operation parameters, a pure virtual method is added. The names and data types of the input parameters for this pure virtual method corresponds to the names and data types of the interface operation parameters
- For each interface attribute parameter of type `CFG`:

  - A 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 attribute are added. 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`:
  - A `const` 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 as they are fixed once and for all [**SpecMetamodel**]

- For each interface attribute of type `DAT`:
  - A 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 as they are modifiable by the component only and not by external entities [**SpecMetamodel**]

In case of the above example:

- `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 `m_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 `constIntegerType&` and one input parameter of type `IntegerType&`
  - getter method for the interface attribute `m_StatusValue` with an input parameter of type `IntegerType&`
  - setter method for the interface attribute `m_StatusValue` with an input parameter of type `constIntegerType&`

Because of the corner case that multiple interfaces can have similar operations, as explained in the previous section, it is necessary to refine these interfaces using the interface helper abstract base classes.

In the above example:

- `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 which has `_InterfaceA` added to the original method signature from the parent interface
- `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 which has `_InterfaceB` added to the original method signature from the parent interface

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

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 to store the call-back functions.

For the above example:

- A struct `operationAddStruct_InterfaceB` is defined
- A struct `statusValueStruct_InterfaceB` is defined

All the infrastructure code entities mentioned above are present in a namespace, named as `General`.

Event emitter ports and event receiver ports

The event emitter port for a particular event is mapped as an abstract base classes and a corresponding concrete implementation class. The event receiver port for a particular event is mapped as an abstract base class.

In case of the above example, the `FailureEventEmitterPort` is mapped as a pair of abstract base class and a concrete implementation class. The abstract base class consists of a function that could be used by external actors to emit a `FailureEvent`.

In case of the above example, the FailureEventReceiverPort is mapped as a pair of abstract base class and a concrete implementation class. The abstract base class consists of a method to receive the FailureEvent and it can be used by an external actor to send an event to the corresponding component.

The FailureEventEmitterPort is present in a namespace, named as Component_Callee and the FailureEventReceiverPort is present in a namespace, named as Component_Caller.

Component types

A component type can be mapped to an abstract base class in C++. A component type must provide all the operations that are listed in the provided interfaces of the component. Hence it inherits from all the interface helper classes which are referenced by its provided interfaces. 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 inherits from the mapped abstract base class for event receiver ports.

A component type must also have pure virtual methods which obtain and release the semaphores on the different operations which it provides. It must also provided pure virtual methods which are necessary to obtain and release semaphores meant for event receptions. In addition to these, pure virtual methods which act as call-back functions for the operations that the component type's required interface ports request with an asynchronous release pattern.

In case of the above example:

- ComponentType in the namespace Component_Caller inherits from the InterfaceA_Helper and also inherits from the abstract base class FailureEventReceiverPort. It has pure virtual methods meant for the purpose of obtaining and releasing of semaphores for the operation CallOperationAdd_InterfaceA and reception of FailureEvent. It also has pure virtual methods which act as call-back functions, for the operation OperationAdd, getter operation for the interface attribute m_StatusValue that the subsumed required interface port RequiredInterfacePortType2 might request with an asynchronous release pattern.
- ComponentType in the namespace Component_Callee inherits from the InterfaceB_Helper. It has pure virtual methods for the purpose of obtaining and releasing of semaphores for operations OperationAdd_InterfaceB, setter and getter operations for the interface attribute m_StatusValue of InterfaceB.

Component Implementations

A component implementation can be mapped in C++ as a concrete implementation of its abstract component type base class. It implements all the pure virtual methods that are inherited from its component type. The 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 kind.

In case of components which promote multiple provided interface ports which refer to the same interface, it is necessary to provide multiple implementations for the operations in the provided interfaces. In order to solve this problem, a component implementation abstract base class is considered. This contains the implementations for the semaphore acquire and release pure virtual methods in the component type class. This component implementation abstract base class is further extended by dummy abstract base classes, one for each of the provided interface ports which refer to the same interface. These dummy abstract base classes are extended by concrete implementation classes of the operations in the corresponding provided interface. As it is a necessity to have only one instantiable concrete implementation per instantiated component, all the concrete implementations are inherited one last time in a component implementation class. Instances of this component implementation are deployable on the hardware platform.

In case of the above example:

- `ComponentImplementation` concrete implementation class in the namespace `Component_Caller` inherits from the abstract base class `Component_Type` in the same namespace. It implements all the pure virtual methods in the `ComponentType` namely:
  - `CallOperationAdd_InterfaceA` inherited from `InterfaceA_Helper` through `ComponentType`
  - Accessing and releasing of semaphore for concurrent access of `CallOperationAdd_InterfaceA`
  - For safe interleaving of concurrent receptions of `FailureEvent`
- Because the component typ

6.4. Mapping of design entities to the infrastructural code

In this section, it is explained how the design entities could be mapped to the infrastructure code that needs to be generated with the model-code transformations through the help of a simple example:

6.4.1. Design entities

Operation parameter structures

As the operations can be called with concurrency kind defined as `deferred`, it is necessary to pack in C++ structures the parameters of the operation (if any) and the address of the component to which the result of the operation (if any), status report of the operation (if any) needs to be sent. This encapsulates all the data necessary to compute the operation, to store the result of the operation and the location to which the result of the operation needs to be sent.

For the above example:

Two operation parameter structures are required namely:

`operationAddStruct` which holds the parameters of the operation `operationAdd` and also the address of the component to which the result of the operation `operationAdd` needs to be sent back

`statusValueStruct` which holds the status value and also the address of the component to which the result of the operation `getStatusValue` needs to be sent

Operation exceptions

As the operations executed can lead to different kind of exceptions, they must be delivered back to the component which called the operation. They can be defined as enums in C++.

For the above example:

As the operation `operationAdd` can raise an exception `OperationAddException`, it is defined as an enum with enum parameters `OperandException, MemoryException, OverflowException, none`

Operation status

The execution status of the operations may be necessary to be delivered to the caller and the statuses are defined as an enums in C++.

For the above example:

The status of the operation `operationAdd` can be sent to the component which makes the call. It is defined as an enum `OperationAddStatus` with enum parameters `Started, Running, Finished`

Operation reports

For a particular operation, the enums of exceptions and status descriptions can be instantiated in a report. These reports are realized as C++ structs.

For the above example:

One operation report is required namely:

`OperationAddReport` holds the instantiation of the enum `OperationAddException` and of the enum `OperationAddStatus`

Call back operations and event receptions

Pure virtual C++ classes are needed for specifying:

- The call back operations for
  - Results of the operations (if any)
  - The statuses of the operations (if any)
- The call back operations for interface attribute getter operations
- Event reception operations

For the above example:

Three pure virtual classes are required namely:

`AsynchronousRequirementsOperationAdd` specifies the call back operation for operation `operationAdd` which consist of the result of the operation available in operation parameter structure `OperationAddStruct` and the report for the operation `OperationAddReport`

`AsynchronousRequirementsStatusValue` specifies the call back operation for getter operation of the status `statusValue`

`AsynchronousRequirementsFailureEventReception` specifies the event reception operation

Component types

Component types are implemented as pure virtual classes in C++. A component type specifies the means for instances of it to connect with other components. They realize the interfaces and also realize the pure virtual classes which specify the call back operations and the event receptions.

For the above example:

There are two component types namely:

`ComponentType_Caller` implements three pure virtual classes namely:

- `InterfaceA`
- `AsynchronousRequirementsOperationAdd` as it calls operation `operationAdd` with concurrency kind `deferred`
- `AsynchronousRequirementsStatusValue` as it calls operation `getStatusValue` with concurrency kind `deferred`
- `AsynchronousRequirementsFailureEventReception` as it receives event `FailureEvent`

`ComponentType_Callee` implements `InterfaceB` only

Component implementations

Component implementations contain the definition of the component type and contains concrete implementations for all the operations in the interfaces it inherits from, for all the call back operations and the event reception operations which were specified in their respective classes. Component implementations are implemented as instantiable C++ classes.

For the above example:

There are two component implementations namely:

`ComponentImplementation_Caller` inherits from the `ComponentType_Caller` and provides concrete implementations for all the classes that it indirectly inherits from

`ComponentImplementation_Callee` A

Chapter 7

Conclusion

Hier bitte einen kurzen Durchgang durch die Arbeit.

Future Work

...und anschließend einen Ausblick

Appendix A

LaTeX-Tipps

A.1. File-Encoding und Unterstützung von Umlauten

Die Vorlage wurde 2010 auf UTF-8 umgestellt. Alle neueren Editoren sollten damit keine Schwierigkeiten haben.

A.2. Zitate

Referenzen werden mittels `\cite[key]` gesetzt. Beispiel: [**WSPA**] oder mit Autorenangabe: **WSPA**.

Der folgende Satz demonstriert 1. die Großschreibung von Autorennamen am Satzanfang, 2. die richtige Zitation unter Verwendung von Autorennamen und der Referenz, 3. dass die Autorennamen ein Hyperlink auf das Literaturverzeichnis sind sowie 4. dass in dem Literaturverzeichnis der Namenspräfix “van der” von “Wil M. P. van der Aalst” steht. **RVvdA2016** präsentieren eine Studie über die Effektivität von Workflow-Management-Systemen.

Der folgende Satz demonstriert, dass man mittels `label` in einem Bibliopgrahie"=Eintrag den Textteil des generierten Labels überschreiben kann, aber das Jahr und die Eindeutigkeit noch von biber generiert wird. Die Apache ODE Engine [**ApacheODE**] ist eine Workflow-Maschine, die BPEL-Prozesse zuverlässig ausführt.

Wörter am besten mittels `\enquote{...}` “einschließen”, dann werden die richtigen Anführungszeichen verwendet.

Listing A.1 lstlisting in einer Listings-Umgebung, damit das Listing durch Balken abgetrennt ist

```
<listing name="second sample">
  <content>not interesting</content>
</listing>
```

Beim Erstellen der Bibtex-Datei wird empfohlen darauf zu achten, dass die DOI aufgeführt wird.

A.3. Mathematische Formeln

Mathematische Formeln kann man so setzen. `symbols-a4.pdf` (zu finden auf http://www.ctan.org/tex-archive/info/symbols/comprehensive/symbols-a4.pdf) enthält eine Liste der unter LaTeX direkt verfügbaren Symbole. Z. B. $\mathbb{N}$ für die Menge der natürlichen Zahlen. Für eine vollständige Dokumentation für mathematischen Formelsatz sollte die Dokumentation zu `amsmath`, ftp://ftp.ams.org/pub/tex/doc/amsmath/ gelesen werden.

Folgende Gleichung erhält keine Nummer, da `\equation*` verwendet wurde.

$$x = y$$

Die Gleichung A.1 erhält eine Nummer:

$$x = y \tag{A.1}$$

Eine ausführliche Anleitung zum Mathematikmodus von LaTeX findet sich in http://www.ctan.org/tex-archive/help/Catalogue/entries/voss-mathmode.html.

A.4. Quellcode

Listing A.1 zeigt, wie man Programmlistings einbindet. Mittels `\lstinputlisting` kann man den Inhalt direkt aus Dateien lesen.

Quellcode im `<listing />` ist auch möglich.

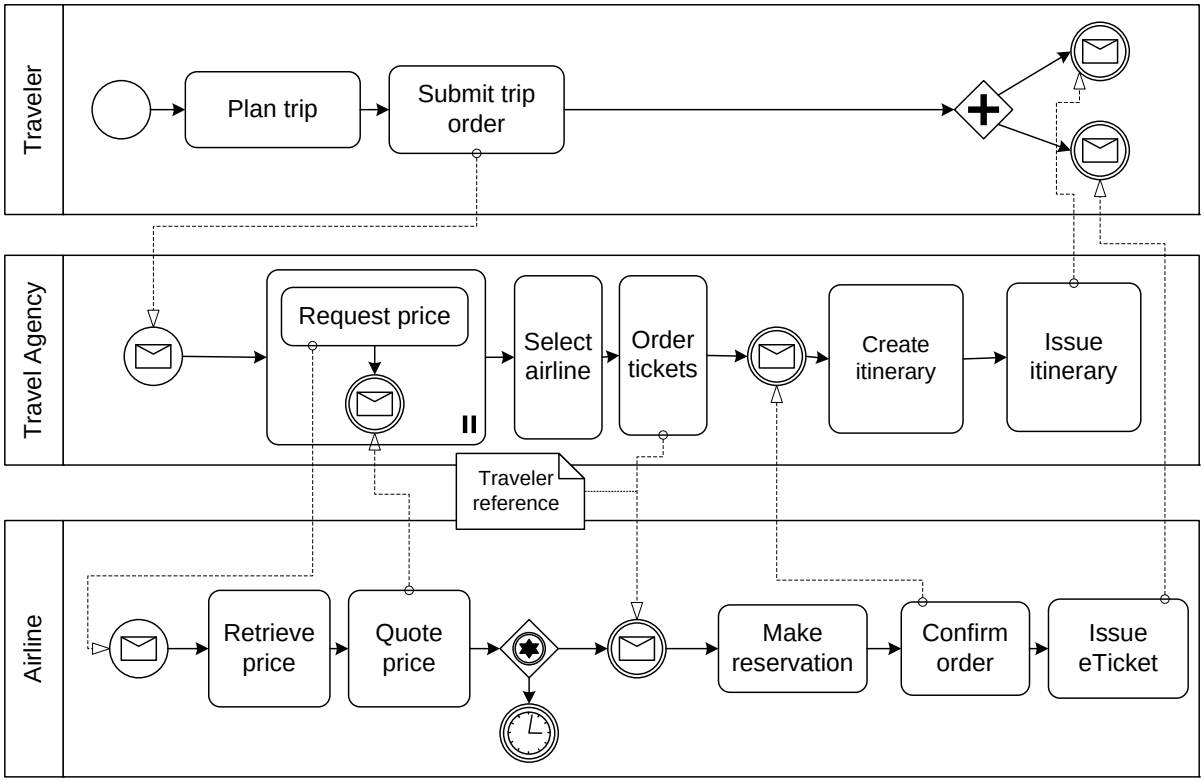


Figure A.1.: Beispiel-Choreographie

A.5. Abbildungen

Die Figure A.1 und A.2 sind für das Verständnis dieses Dokuments wichtig. Im Anhang zeigt Figure A.4 on page 67 erneut die komplette Choreographie.

Das SVG in **??** ist direkt eingebunden, während der Text im SVG in **??** mittels pdflatex gesetzt ist.

Falls man die Graphiken sehen möchte, muss inkscape im PATH sein und im Tex-Quelltext `\iffalse` und `\iftrue` auskommentiert sein.

A.6. Tabellen

Table A.1 zeigt Ergebnisse und die Table A.1 zeigt wie numerische Daten in einer Tabelle representiert werden können.

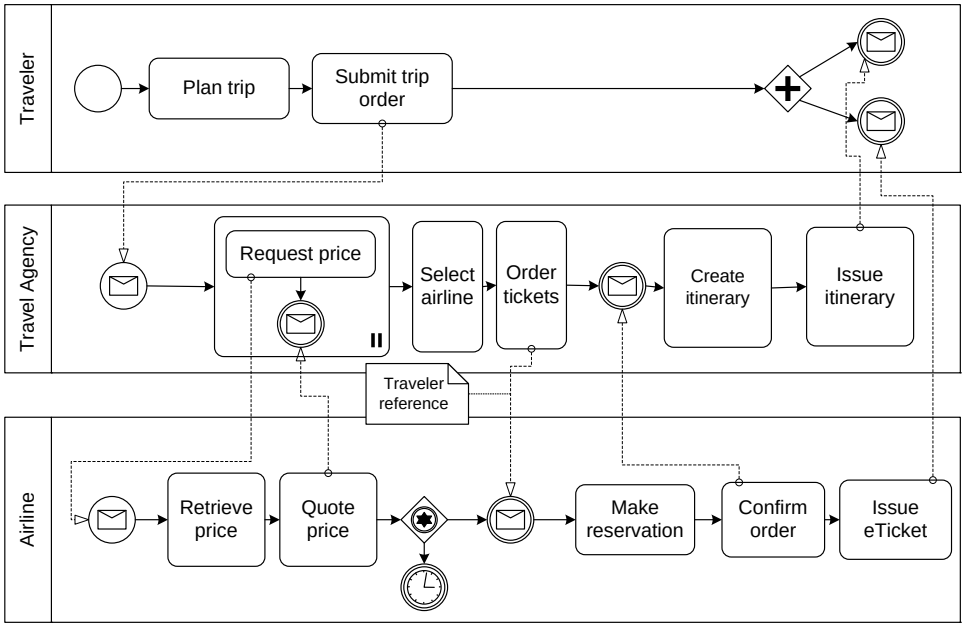
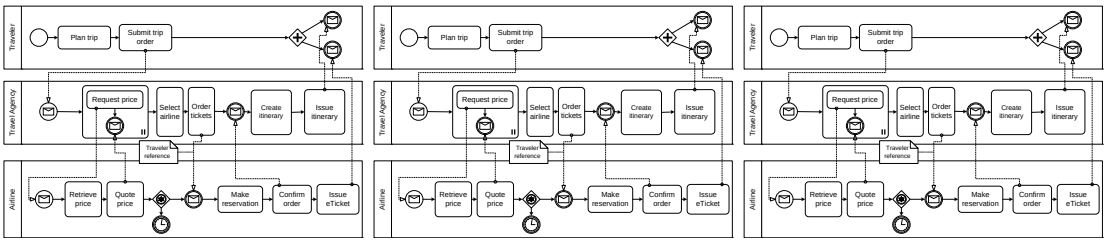


Figure A.2.: Die Beispiel-Choreographie. Nun etwas kleiner, damit `\textwidth` demonstriert wird. Und auch die Verwendung von alternativen Bildunterschriften für das Verzeichnis der Abbildungen. Letzteres ist allerdings nur Bedingt zu empfehlen, denn wer liest schon so viel Text unter einem Bild? Oder ist es einfach nur Stilsache?



(a) **(b)** **(c)** Subcaption if needed

Figure A.3.: Beispiel um 3 Abbildung nebeneinader zu stellen nur jedes einzeln referenzieren zu können. Abbildung A.3b ist die mittlere Abbildung.

zusammengefasst		Titel
Tabelle	wie	in
tabsatz.pdf	empfohlen	gesetzt
Beispiel	ein schönes Beispiel für die Verwendung von “multirow”	

Table A.1.: Beispieltabelle – siehe http://www.ctan.org/tex-archive/info/german/tabsatz/

	Parameter 1		Parameter 2		Parameter 3		Parameter 4	
Bedingungen	**M**	**SD**	**M**	**SD**	**M**	**SD**	**M**	**SD**
W	1.1	5.55	6.66	.01				
X	22.22	0.0	77.5	.1				
Y	333.3	.1	11.11	.05				
Z	4444.44	77.77	14.06	.3				

Table A.2.: Beispieltabelle für 4 Bedingungen (W-Z) mit jeweils 4 Parameters mit (M und SD). Hinweiß: immer die selbe anzahl an Nachkommastellen angeben.

A.7. Pseudocode

Algorithm A.1 zeigt einen Beispielalgorithmus.

Algorithmus A.1 Sample algorithm

procedure SAMPLE(a,v_e)
  parentHandled $\leftarrow (a = \mathsf{process}) \vee \mathsf{visited}(a'), (a', c, a) \in \mathsf{HR}$
    // $(a', c'a) \in \mathsf{HR}$ denotes that a' is the parent of a
  if parentHandled $\wedge (\mathcal{L}_{in}(a) = \emptyset \vee \forall l \in \mathcal{L}_{in}(a) : \mathsf{visited}(l))$ **then**
    $\mathsf{visited}(a) \leftarrow$ true
    $$\mathsf{writes}_\circ(a, v_e) \leftarrow \begin{cases} \mathsf{joinLinks}(a, v_e) & |\mathcal{L}_{in}(a)| > 0 \\ \mathsf{writes}_\circ(p, v_e) & \exists p : (p, c, a) \in \mathsf{HR} \\ (\emptyset, \emptyset, \emptyset, false) & \text{otherwise} \end{cases}$$
    if $a \in \mathcal{A}_{basic}$ **then**
      HANDLEBASICACTIVITY(a,v_e)
    else if $a \in \mathcal{A}_{flow}$ **then**
      HANDLEFLOW(a,v_e)
    else if $a = \mathsf{process}$ **then** // Directly handle the contained activity
      HANDLEACTIVITY(a',v_e), $(a, \perp, a') \in \mathsf{HR}$
      $\mathsf{writes}_\bullet(a) \leftarrow \mathsf{writes}_\bullet(a')$
    end if
    for all $l \in \mathcal{L}_{out}(a)$ **do**
      HANDLELINK(l,v_e)
    end for
  end if
end procedure

Und wer einen Algorithmus schreiben möchte, der über mehrere Seiten geht, der kann das nur mit folgendem **üblen** Hack tun:

Algorithmus A.2 Description

code goes here
test2

A.8. Abkürzungen

Beim ersten Durchlaf betrug die Fehlerrate (FR) 5. Beim zweiten Durchlauf war die FR 3.

Mit `\ac{...}` können Abkürungen eingebaut werden, beim ersten aufrufen wird die lange Form eingesetzt. Beim wiederholten Verwenden von `\ac{...}` wird automatisch die kurz Form angezeigt. Außerdem wird die Abkürzung automatisch in die Abkürzungsliste eingefügt.

Definiert werden Abkürzungen in der Datei *ausarbeitung.tex* im Abschnitt '%%% acro' mithilfe von `\DeclareAcronym{...}{...}`.

Mehr infos unter: http://mirror.hmc.edu/ctan/macros/latex/contrib/acro/acro_en.pdf

A.9. Verweise

Für weit entfernte Abschnitte ist "varioref" zu empfehlen: "Siehe Appendix A.3 on page 60". Das Kommando `\vref` funktioniert ähnlich wie `\cref` mit dem Unterschied, dass zusätzlich ein Verweis auf die Seite hinzugefügt wird. `vref`: "Appendix A.1 on page 59", `cref`: "Appendix A.1", `ref`: "A.1".

Falls "varioref" Schwierigkeiten macht, dann kann man stattdessen "cref" verwenden. Dies erzeugt auch das Wort "Abschnitt" automatisch: Appendix A.3. Das geht auch für Abbildungen usw. Im Englischen bitte `\Cref{...}` (mit großen "C" am Anfang) verwenden.

A.10. Definitionen

Definition A.10.1 (Title)
Definition Text

Definition A.10.1 zeigt ...

A.11. Verschiedenes

KAPITÄLCHEN werden schön gesperrt...

I. Man kann auch die Nummerierung dank paralist kompakt halten
II. und auf eine andere Nummerierung umstellen

A.12. Weitere Illustrationen

Abbildungen A.4 und A.5 zeigen zwei Choreographien, die den Sachverhalt weiter erläutern sollen. Die zweite Abbildung ist um 90 Grad gedreht, um das Paket `rotating` zu demonstrieren.

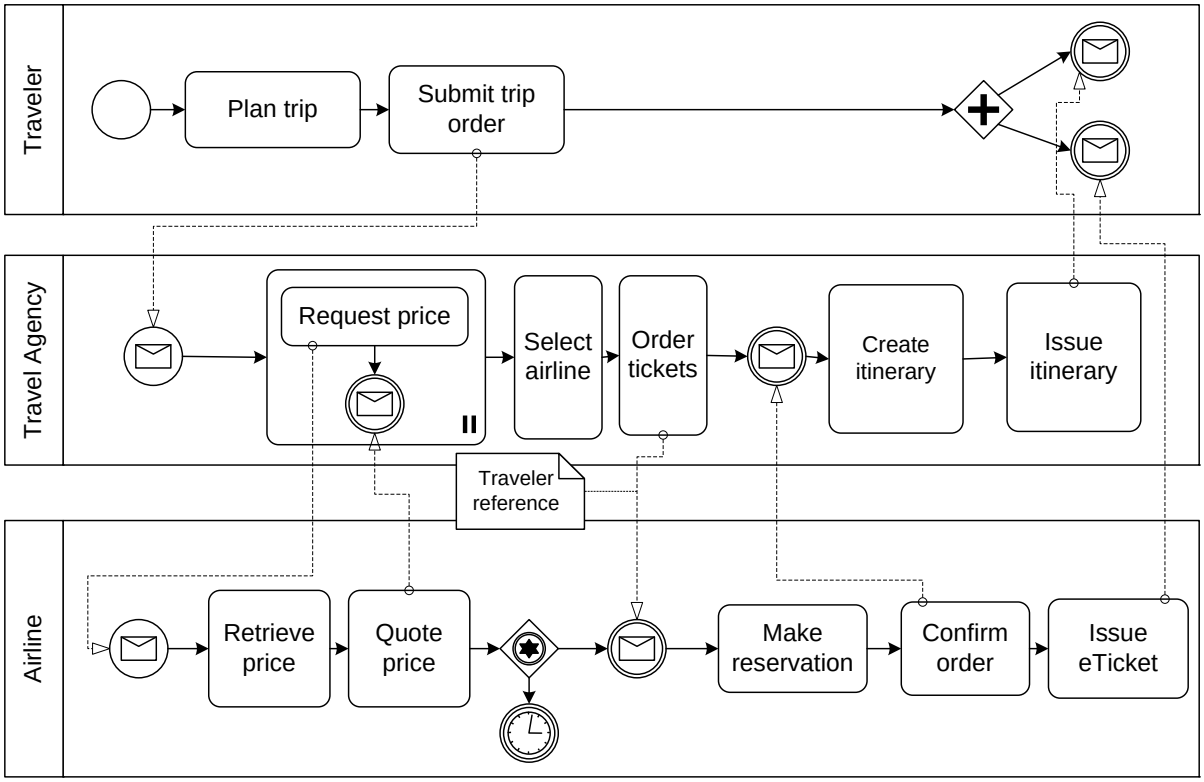


Figure A.4.: Beispiel-Choreographie I

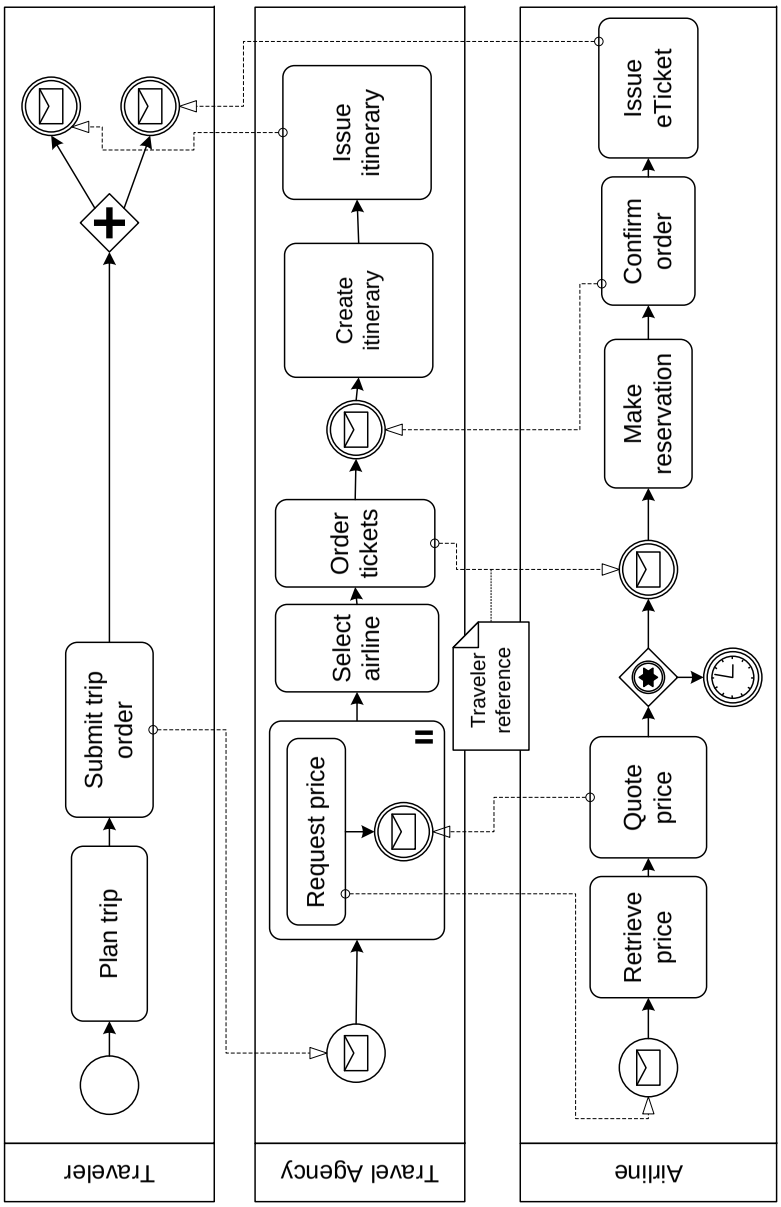


Figure A.5.: Beispiel-Choreographie II

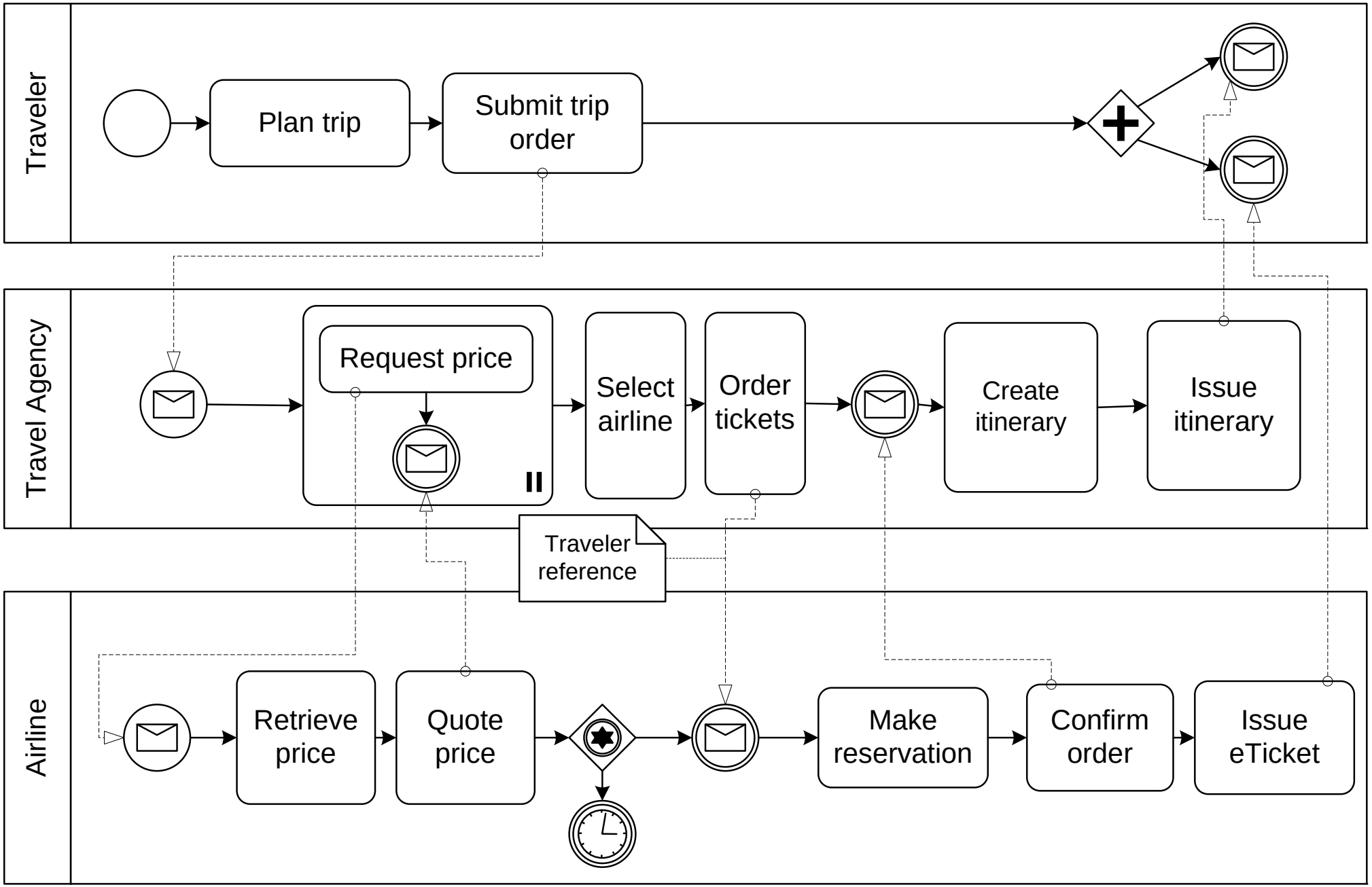


Figure A.6.: Beispiel-Choreographie, auf einer weißen Seite gezeigt wird und über die definierten Seitenränder herausragt

A.13. Schlusswort

Verbesserungsvorschläge für diese Vorlage sind immer willkommen. Bitte bei github ein Ticket eintragen (https://github.com/latextemplates/uni-stuttgart-computer-science-template/issues).

All links were last followed on March 17, 2008.

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