Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Sapce 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**] [AS10] 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 6 – 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 already for quite some time 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 [AS10]. The implementation of such a vision is expected to provide benefits for all the stake-holders in the space community [AS10].

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 [HG10]. 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 [SF11]

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 [AS10]

ESA through its two parallel activities, namely COrDeT and DOMENG [RFA+12] 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 on-board software reference architectures called "SAVOIR Fair Architecture and Interface Reference Elaboration" working group (SAVOIR FAIRE). OSRA is the result of the R&D activities of this group [AS10].

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 [AS10]. 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 [07], 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)[Kru00], 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 con- texts, 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 system 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 [Pan11] [PV13b].

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 [AS10].

A software reference architecture is made up of two main parts: [AS10]

- 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

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 [AS10]. 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 [Pan11]. 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*"[Gmb]

The OBSW life-cycle needs to be consistent with the system life-cycle, which features the definition of functional increments in system development [AS10]. 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 [RFA+12][AS10]. These stakes are included and defined as the user needs [AS10] [Pan11] for the development of OSRA:

Shorter software development time Need for faster software development in the context of a shorter schedule.

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 [AS10] [Pan11].

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 [PV09]. 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 [PV09]. 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 [PV14] and the ESA: reference Earth Observation case study [PV14].

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 my 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 [Gén].

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: [RFA+12] [PV13b] [AS10]

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

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 [BPV08]

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.[PV13a]

A conforming Execution Platform An execution platform helps to preserve at run-time, 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 [CLWK00]. 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 obvious as in the other main-stream domains like

enterprise computing [AS10] because of strong verification and validation requirements imposed in the space domain and the presence of non-functional dimensions [AS10].

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 [AS10]:

- 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 [PV09]

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 [PV14]. 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 [PV14] [Pan11]. 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) [PV14] wherein:

- The components are designed
- The products designed by the design environment can be verified and analyzed by the design environment.
- The lower lever 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 [PV14] and it helps to separate the aspects of software design and implementation. OSRA and its associated component model promotes separation of concerns[PV14][Pan11]:

- 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 [BPV08].
- 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 [PV13a].

Inculcating the principle of separation of concerns in the development process has two major benefits [PV14]:

- It increases the reuse potential of the components, which is an important high level requirement described in the previous section [AS10]. 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 [PV14]. 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 [AS10][PV]. 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 [PV14].

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 [Pan11].

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" [PV14] [Fow]. 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 [PV09]. 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. The former principle is called interface "promotion" and the latter is called the interface "subsumption" [PV09]. 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 anything 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 component 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, as the nature of the target system chosen, which is a simple linux based system, 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 [AS10]. 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 [BPV08]. 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 [AS10]:

- Pollution of the user-models with entities that are more primitive and is 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 proving 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 [AS10]:

Services for containers These services are meant to be used by the containers eg. 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 [Pan11]:

- 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 valid prediction of the system at at run-time [Pan11].

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 [PV14]. 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, it may indicate some shortcomings and flaws in the design of the component. That is when a re-design, re-negotiation of the component definition needs to be done and it might often lead to an iterative/incremental development process [BPV08]. Detailed design and implementation of components is 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 the architecture: 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 [AS10][PV14]. 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 [PV14] [Pan11] [AS10]:

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 [PV14]. 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.

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 thence a particularly attractive unit to be subcontracted to third-party because the software architect can define components, attach technical budgets to it and delegate the implementation to he third parties. The third party might add additional operations to the component implementation as and when necessary for the implementation [PV14].

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 help 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 [PV14]. At first, the concurrency kind of the operation is established, and they can be `immediate` or `deferred` operations. In case of `immediate` operation, it is executed in the flow of control of the caller (synchronous) and in case of `deferred` operation (asynchronous), 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 `deferred`, 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

Please note that, this step is not of concern in this Master thesis, as it deals deals with hardware modeling which is outside the current scope. However, this step was mentioned for sake of continuity and clarity.

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 [PV14]. 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.

Note: As the model based analysis of the user-model requires a model representation of the containers and connectors, and as the main concern of this master thesis is to generate containers and connectors amongst many others, it is not possible to do an accurate model based schedulability analysis as it is outside the scope of the Master thesis

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 [AS10] [Pan11] [PV14]. The design flow is as explained in the previous section.

One of the advantages of the design views is to promote or enforce a certain design flow [PV14]. 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 [Pan17]. 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 [EO16].

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 [EO16].

Chapter 4

Tasking Framework

Chapter 5

Infrastructural code generation

5.1. Introduction

In the previous chapters we have seen model-driven software development approach that was centered on component-based techniques. Dijkstra's principle of separation of concerns was one of the corner principles which was part of the software reference architecture and the component model proposed. According to it the user design space should be limited to the internals of the components where only strictly sequential code needs to be used and the extra non-functional requirements are declaratively specified in the form of annotations on the component provided interfaces.

As discussed before in the previous chapters, the reference software architecture was found to be made up of a component model, computational model, programming model and a conforming execution platform. It was also understood that the component model should be statically bind to a computational model to formally define the computational entities, as well as the rules which govern their usage.

A reference programming model was developed for this purpose and it was found that the Ada Ravenscar profile (RP) was found to be particularly attractive to help realize this goal as it does not allow all language constructs that are exposed to unbounded execution-time and non-determinism and RP was the backbone of the programming model. A set of code archetypes were also developed in accordance with this programming model.

The component based approach was put into trial in two parallel and complementary efforts: Development of on-board software under European Space Agency (ESA) initiatives and in the CHESS project, which targets space, telecom and railway applications.

For the realization of extra functional properties or more precisely for the generation of the complete infrastructure code consisting of two parts namely:

- Automated generation of the non-functional code, plus the interface code and the skeletons for the components
- Automated generation of component containers and component connectors

a code generator is used and thence the third-party user has to solely implement the functional code of the components. This is in line with the principle of separation of concerns.

The ASSERT project (Automated proof-based System and Software Engineering for Real-Time systems), was the first, large project which pursued this vision and showed the feasibility of a development approach for high-integrity real-time systems centered on separation of concerns, correctness-by-construction and property preservation.

The code archetypes, which were one of the outcome of the ASSERT project complemented the Ada Ravenscar Profile, which used Ada run-time, which is more compact in foot-print and hence could fit the needs of typical embedded systems which were resource-constrained. The code archetypes were also amenable to automated code generation and were an evolution of previous work on code generation from HRT-HOOD to Ada.

These code archetypes form the very basis of the programming model which is developed in this master thesis.

5.2. 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:

5.2.1. Problem description

A very simple OBSW model consists of two components, namely `ComponentA` and `ComponentB`

`ComponentA` has the following requirements:

- The function `startOperation` implemented in the component implementation needs to be called on the periodic basis with a period of two seconds.
- There should be two required interface ports.

  - First required interface port allows `ComponentA` to call an operation named `operationAdd`, set and get the status value `statusValue` with concurrency kind `immediate`
  - Second required interface port allows `ComponentA` to call the same operation `operationAdd`, allows to set and get the same status value `statusValue` with concurrency kind as `deffered`.
- Can receive messages or events asynchronously and there should be a event receiver port which listens to a particular event called the `FailureEvent`
- Can receive a report `OperationAddReport` for the operation `operationAdd` which is called. The report has status of the operation `operationAdd` called the `operationAddStatus` and the exception for the operation `operationAdd` called the `OperationAddException`

`ComponentB` has the following requirements:

- Provides two component implementations. Both the implementations provides their own
  - Versions of the operation `operationAdd` which can be called by any other components
  - Instances of the status value `statusValue` which can be written to and read from other components
- Two provided interface ports and they have non-functional requirements attached to them
  - First provided interface port works with the first version of component implementation and it requires `operationAdd` to be a `protected` operation and needs the status value `statusValue` to be set and get in an unprotected way
  - Second provided interface port works with the second version of component implementation and it requires `operationAdd` to be called in a sporadic way with a `Minimum Inter-Arrival Time (MIAT)` of two seconds and needs the status value `statusValue` to be set and get in an protected way.
- Can send messages or events asynchronously and there should be a event emitter port which emits a particular event called the `FailureEvent`
- Can send report report `OperationAddReport` for the operation `operationAdd` which is called. The report has status of the operation `operationAdd` called the `operationAddStatus` and the exception for the operation `operationAdd` called the `OperationAddException`

From the above description it is clear that these components can be connected to each other or in particular the first required interface port can be connected to the first provided interface port and the second required interface port can be connected to the second provided interface port. The following different design entities are thereby constructed:

5.2.2. Design entities

Interfaces

Interfaces are implemented as C++ pure virtual classes and they have entries for the interface operations, interface attribute access operations and interface attributes. Concrete implementations for the interface operations and the interface attribute access operations need to be provided by the classes that implement these interfaces

For the above example:

Two primary interfaces can be created based on the problem description namely:

`InterfaceA` This specifies a single operation `startOperation`

`InterfaceB` This specifies

- The operation `operationAdd`
- Actual status value `statusValue`
- Getter operation for the status value `statusValue` called `getStatusValue`
- Setter operation for the status value `statusValue` called `setStatusValue`

Three secondary interfaces for `InterfaceB` are created:

`InterfaceB_Synchronous` which redefines the interface `InterfaceB` for cases where the operations in the interface `InterfaceB` are called synchronously

`InterfaceB_Asynchronous` which redefines the interface `InterfaceB` for cases where the operations in the interface `InterfaceB` are called asynchronously

`InterfaceB_extension` which inherits the interface `InterfaceB` to add additional operations to handle the situation when the operations in the interface `InterfaceB` are called asynchronously

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 6

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.

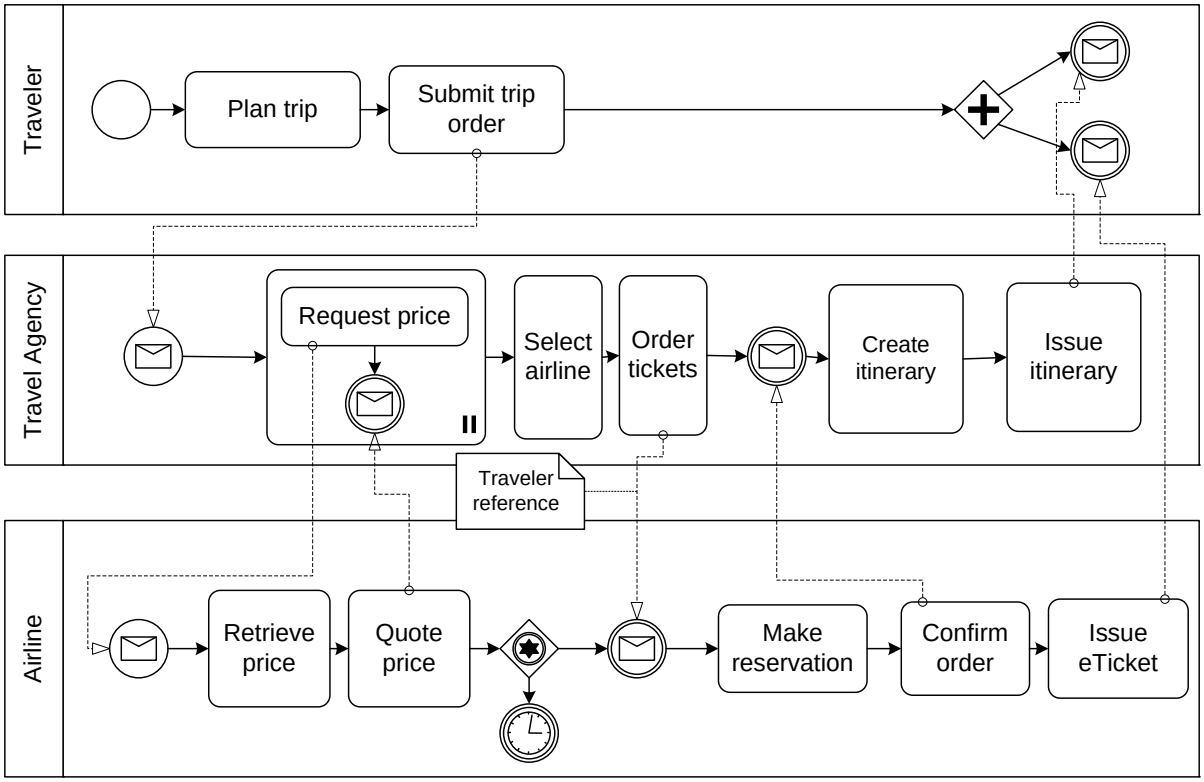


Figure A.1.: Beispiel-Choreographie

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 49 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.

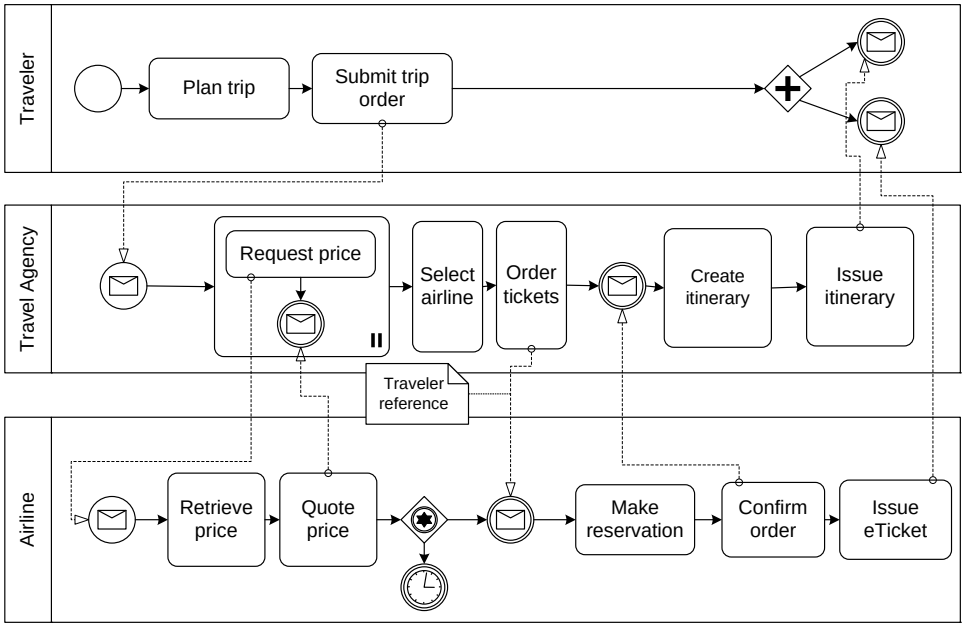
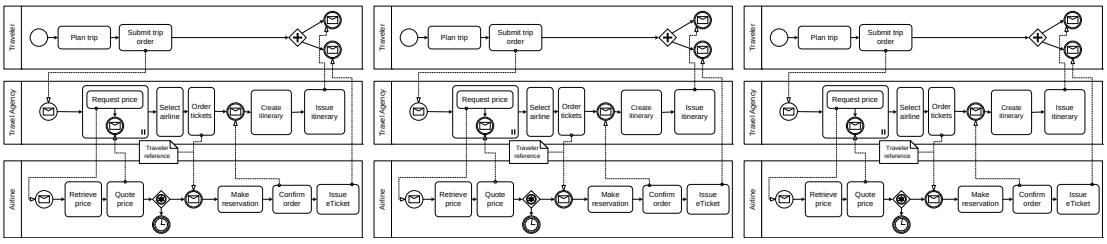


Figure A.2.: Die Beispiel-Choreographie. Nun etwas kleiner, damit `\textwidth` demonstriert wird. Und auch die Verwendung von alternativen Bildunterschriften für das Verzeichnis der Abbildungen. Letzteres ist allerdings nur Bedingt zu empfehlen, denn wer liest schon so viel Text unter einem Bild? Oder 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 42". 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 41", `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.

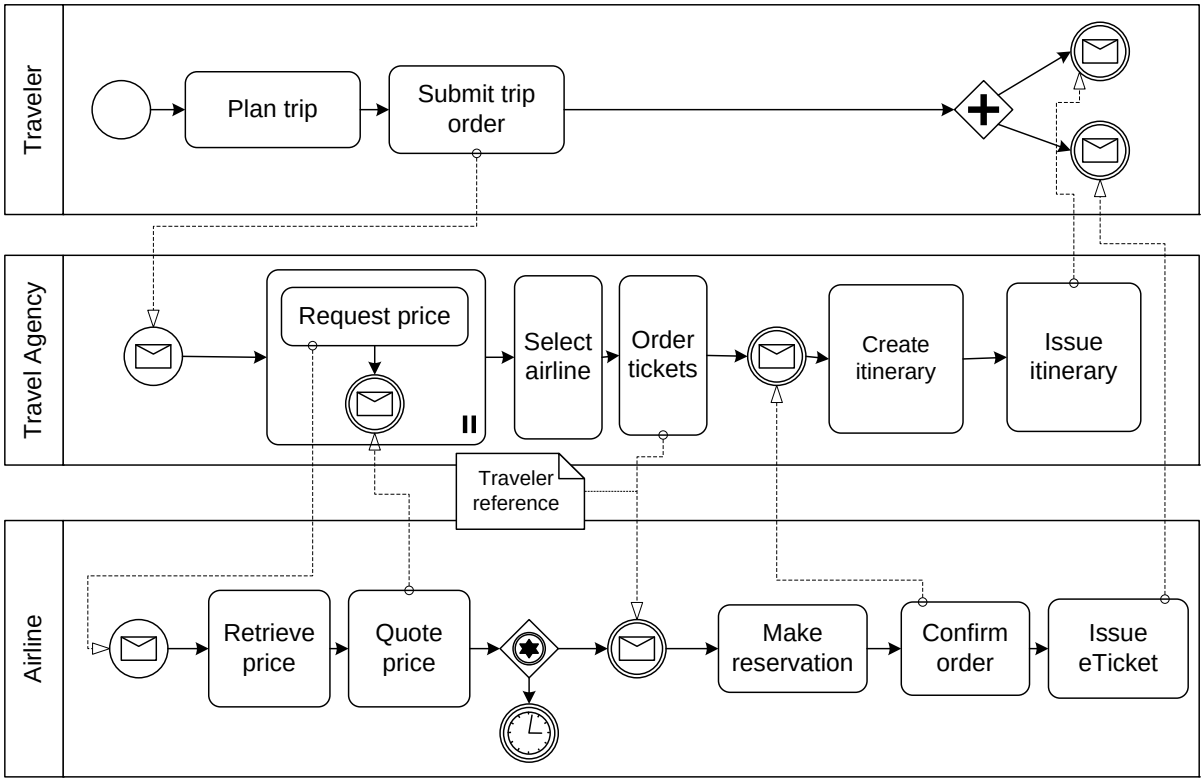


Figure A.4.: Beispiel-Choreographie I

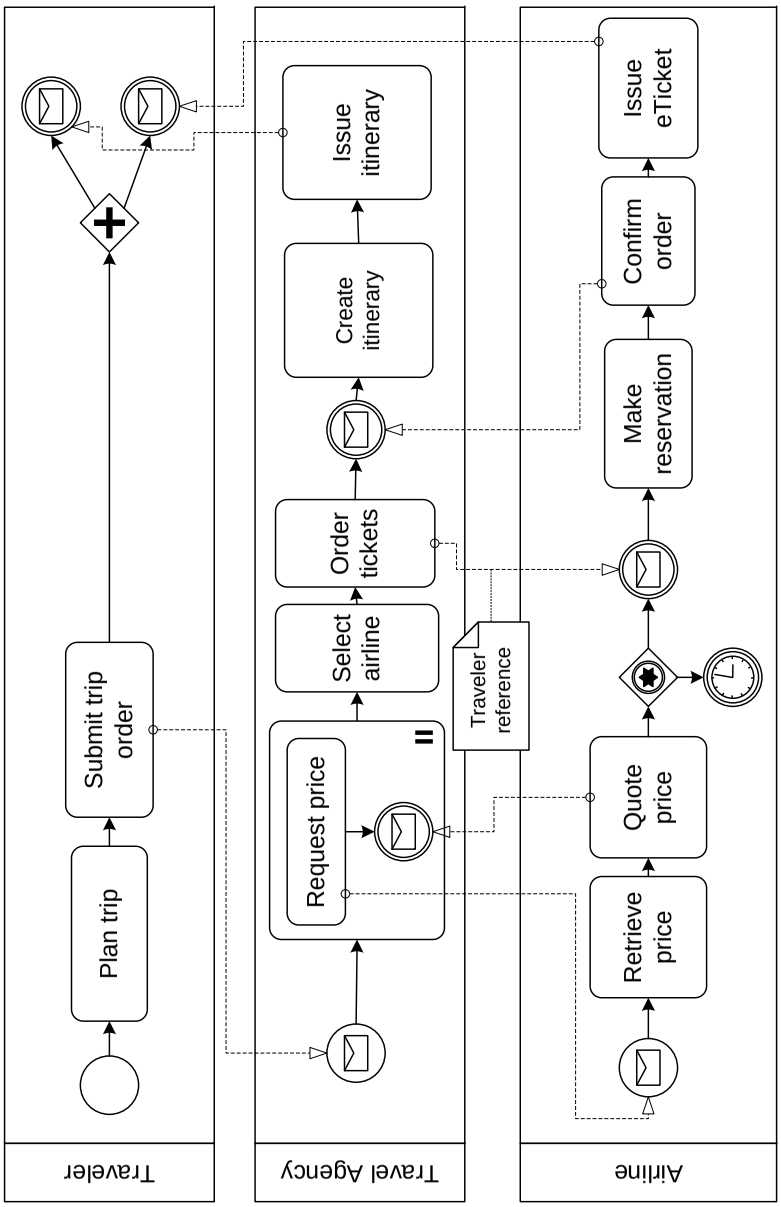


Figure A.5.: Beispiel-Choreographie II

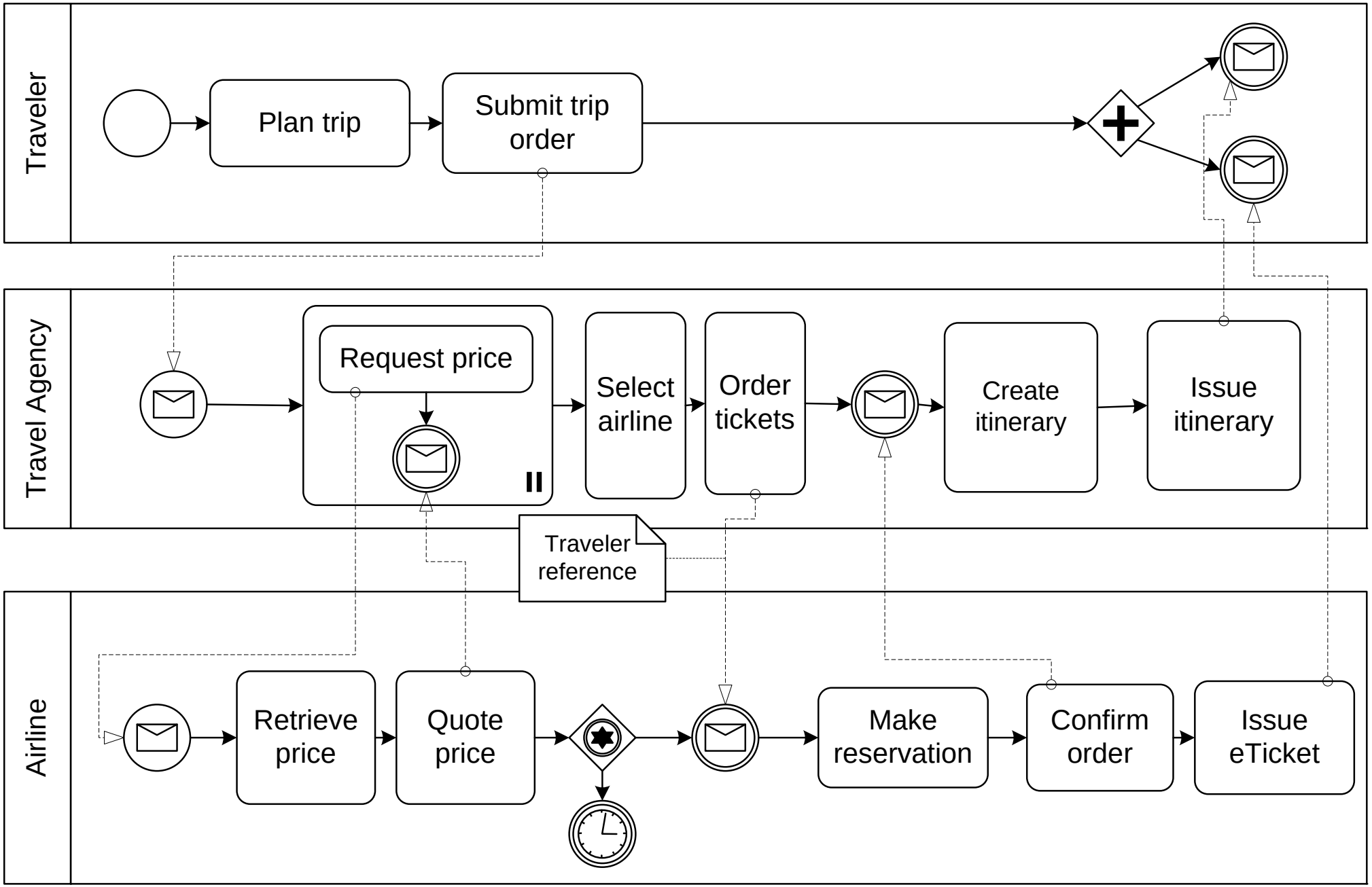


Figure A.6.: Beispiel-Choreographie, auf einer weißen Seite gezeigt wird und über die definierten Seitenränder herausragt

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).

Bibliography

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

[AS10] M. P. Andreas Jung, J.-L. T. supported by the Savoir-Faire working group. *SAVOIR-FAIRE - On-board Software Reference Architecture*. Tech. rep. ESTEC-European Space Technology and Research Centre, 2010 (cit. on pp. 1, 3–8, 10–13, 15, 16, 20, 25).

[BPV08] M. Bordin, M. Panunzio, T. Vardanega. "Fitting Schedulability Analysis Theory into Model-Driven Engineering." In: *2008 Euromicro Conference on Real-Time Systems*. July 2008, pp. 135–144. DOI: 10.1109/ECRTS.2008.12 (cit. on pp. 10, 12, 15, 19).

[CLWK00] X. Cai, M. R. Lyu, K.-F. Wong, R. Ko. "Component-based software engineering: technologies, development frameworks, and quality assurance schemes." In: *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*. 2000, pp. 372–379. DOI: 10.1109/APSEC.2000.896722 (cit. on p. 10).

[EO16] ESA, Obeo. *User Manual of the OSRA SCM Model Editor*. Tech. rep. European Space Agency, 2016 (cit. on pp. 26, 27).

[Fow] M. Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. URL: https://martinfowler.com/articles/injection.html (cit. on p. 14).

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

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

[HG10] W. D. H. Bo D. Hui, Z. Guifan. “Basic Concepts on AUTOSAR Development.” In: *2010 International Conference on Intelligent Computation Technology and Automation*. (Changsha, China, May 11, 2010). IEEE, 2010, pp. 871–873. ISBN: 978-1-4244-7280-2. DOI: 10.1109/ICICTA.2010.571 (cit. on p. 3).

[Kru00] P. Kruchten. *The Rational Unified Process: An Introduction – 2nd Edition*. Addison-Wesley Professional, 2000. ISBN: 0201707101 (cit. on p. 5).

[Pan11] M. Panunzio. “Definition, realization and evaluation of a software reference architecture for use in space applications.” PhD thesis. Universitàdi Bologna Università degli Studi di Padova, 2011 (cit. on pp. 5–8, 11, 12, 14, 16, 17, 20, 25).

[Pan17] M. Panunzio. *Specification of the Metamodel for the OSRA Component Model*. Tech. rep. Thales Alenia Space - France, 2017 (cit. on p. 25).

[PV] M. Panunzio, T. Vardanega. “A Component Model for On-board Software Applications.” In: *36th EUROMICRO Conference on Software Engineering and Advanced Applications*. (Sept. 1, 2010). Lille, France: IEEE, pp. 57–64. ISBN: 978-1-4244-7901-6. DOI: 10.1109/SEAA.2010.39 (cit. on pp. 10, 13).

[PV09] M. Panunzio, T. Vardanega. “On Component-Based Development and High-Integrity Real-Time Systems.” In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2009, pp. 79–84. DOI: 10.1109/RTCSA.2009.15 (cit. on pp. 8, 9, 11, 14).

[PV13a] M. Panunzio, T. Vardanega. “Charting the Evolution of the Ada Ravenscar Code Archetypes.” In: *Ada Lett.* 33.1 (June 2013), pp. 64–83. ISSN: 1094-3641. DOI: 10.1145/2492312.2492320. URL: http://doi.acm.org/10.1145/2492312.2492320 (cit. on pp. 10, 12).

[PV13b] M. Panunzio, T. Vardanega. “On Software Reference Architectures and Their Application to the Space Domain.” In: *Safe and Secure Software Reuse*. Ed. by J. Favaro, M. Morisio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–159. ISBN: 978-3-642-38977-1 (cit. on pp. 5, 10).

[PV14] M. Panunzio, T. Vardanega. “A component-based process with separation of concerns for the development of embedded real-time software systems.” In: *Journal of Systems and Software* 96 (2014), pp. 105–121. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.05.076. URL: http://www.sciencedirect.com/science/article/pii/S0164121214001381 (cit. on pp. 10–14, 19–23, 25).

[RFA+12] A.-I. Rodriquez, F. Ferrero, E. Alana, A. Jung, M. Panunzio, T. Vardanega, A. Grenham. “The Component Layer of Cordet On-Board Software Architecture.” In: *DASIA 2012 Data Systems In Aerospace*. (in Drubrovnik, Croatia, May 14, 2012). Ed. by L. O. E. SP-701. 2012. ISBN: 978-92-9092-265-0 (cit. on pp. 4, 7, 10).

[SF11] J. A. S. Dersten, J. Froberg. “Effect Analysis of the Introduction of AUTOSAR - A Systematic Literature Review.” In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. (Oulu, Finland, Aug. 30, 2011). IEEE, 2011, pp. 239–246. ISBN: 978-1-4577-1027-8. DOI: 10.1109/SEAA.2011.44 (cit. on p. 3).

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