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 Kapitel steht die Einleitung zu dieser Arbeit. Sie soll nur als Beispiel dienen und hat nichts mit dem Buch [**WSPA**] 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 standardisation in the avionics system in order to increase the efficiency and reduce cost and schedule in the development. The implementation of such a vision is expected to provide benefits for all the stake-holders in the space community:

Customer Agencies Significant drop in the project development lifecycle and the risk involved in the software development

System Integrators There would be increased competition amongst them to deliver at lower price and maintain shorter time-to market and there would also be multi-supplier option

Supplier Industry Benefits from diversified customer bases and the supplied building blocks would be compatible with prime architectures across the board

Similar initiatives have already been taken across various industries and eg. AUTOSAR for the automotive industry is worthy mentioning. Space can benefit from these examples by studies related to how these similar initiatives were successfully conducted and how they fared. ALthough the business model is different in the automotive and the space sectors, AUTOSAR demonstrates the need fir standardization is the key irrespective of the sector and is driven by the need of the industry to become more competitive.

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

ESA through its two parallel activities aimed at increasing the software reuse in on-board softwares (CORDET and Domeng) 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 R&D activities of this group.

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. 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.

The 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. Elevating a software architecture to 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.

2.2. Need for reference architecture

2.2.1. Motivation

The schedules of space projects are always decreasing and the team need to increase their efficiency and cost effectiveness in the development process of on-board avionics. But the on-board software is getting more complex because of the trend towards more functionality being implemented by the on-board software. Therefore the overall

objective of space industry is now 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 fully compatible building blocks, plus specific adaptations and "missionisation" according to specific mission requirements. The target missions are the core ESA missions, ie.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.

Usually a software building block:

- 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. 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"

2.2.2. Software reference architecture

The reference architecture is made up of two main parts:

- An 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

2.2.3. User needs

These are some of the needs that were assimilated to guide the development of the software reference architecture:

Shorter software development time The software development schedule should be reduced because usually the definition of the software requirements is done at a later stage and the final version of the software is expected to be released earlier. Even though the cost of the software itself us a minor fraction of the cost of the whole system in space industry, the impact of delays in availability of the software may have a huge impact on the overall schedule and consequently on the cost of the project

Reduce recurring costs It is important to identify and reduce the recurring costs and in turn help to use the project resources to focus on value added to the product or to reduce the cost of development while providing the same set of functions. Examples for recurring costs include device drivers, real-time operating system, providing communication services etc and it is important to note that these cots drivers do not provide an added value and are not mission specific.

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

Increase cost-efficiency Cost-efficiency is the "value" of the software product that is developed with a certain amount of budget. An increased cost-efficiency is achieved by developing the same set of functions for less budget, developing the same set of functions with more stringent requirements for the same budget and increasing the number of realized functions for the same budget. The budget available for the software development is not expected to grow and it may be indeed be subjected to reduction and hence new development approaches may be required to fulfill

this user need. On contrary the performance of the application building blocks eg. accuracy of the AOCS controls is expected to grow and new complex functionalities are expected to develop

Reduce Verfication and Validation effort The main contributor to the cost of software development are V&V activities which contribute anywhere between 50% to 70% of the overall cost. Adoption of the principle of Correction by Construction (C-by-C) which is one of the founding principles of choice (refer Chapter 2), analysis at early design stage and provisions for re-usability of (functional) tests are expected to reduce the Verification and Validation efforts. This also leads to shorter software development times and reduced costs.

Mitigate the impact of late requirement definition or change Late refinement of system design, evolution of the operational level, late finalization of the system FDIR, software modification to compensate the problems in the hardware found during system integration may often lead to definition of new requirements or their changes for the software, anytime in the entire SW life-cycle.

Support for various system integration strategies Preliminary software releases are important to allow early system integration and software development may be managed with different strategies. It is necessary to respect these strategies and help final integration of increments or elements.

Simplification and harmonization of FDIR Simplification and coordination of the Fault Detection, Isolation and Recovery (FDIR) needs to be handled at both the system and software level. System engineers and software implementers need to justify the definition of the FDIR strategy at the system level and the software level respectively. A set of functionalities and design patterns need to be provided at the software level that cater to necessary mechanisms for the software realization of FDIR strategy.

Optimize flight maintenance Flight maintenance may be required to change the OBSW and provision of the required operations and coordination of the strategy to perform it will decrese the time and cost of maintenance. It is better if parts of the software could be updated without having to reboot the CDMU.

Industrial policy support The development process should enable multi-team software development. It is necessary to incorporate certain flexibility in the allocation of the software elements to industry, according to certain criterion such as prime/non-prime, or geographical return. Multi-team software development is essential to subcontract to non-primes while be in charge of the integration and apply the geographical return policy.

Role of software suppliers As discussed before, the new approach must increase the competence of the supplier and foster competition amongst the suppliers: Different suppliers may develop the same component and compete on quality, extension features, performance, cost and schedule. The suppliers will also profit from this approach as they do not have to adapt the software to specific development policies if each single prime.

Dissemination activities System engineers can be exposed to core principle of the process and if they derive specifications for the system out of the domain of reuse, the costs will certainly increase.

Future needs The trend of increasing complexity of the OBSW gives rise to several needs and these needs need to be subjected to evaluation and their impact on the software reference architecture needs to be monitored. Some of the examples of the future needs include integration of functions of different criticality and security levels, use of Time and Space Partitioning (TSP), support to the multi-core processors, contextual verification of safety properties.

2.2.4. High level requirements

The user needs were translated into a set of high-level requirements:

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 the unit, integration and validation tests by providing a pre-qualification data package supported by a SW Reuse File in the sense of ECSS software standards. Traced to user needs:

- Shorter software development time
- Reduce recurring costs

Separation of concerns Separation of concerns is one of the cornerstone principles and it deals wit 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 and hence increasing the software reuse. Traced to user needs:

- Quality of the product
- Reduce Verification and Validation effort
- Role of software suppliers

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. Traced to user needs:

- Shorter software development time
- Reduce Verification and Validation effort

HW/SW Independence It enables development of the software independent from the hardware features. Its 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 isolated from the changes in the hardware-dependent layer. Traced to user-needs:

- Quality of the product
- Mitigate the impact of later requirements definition or change
- Support for various system integration changes

Component based approach The whole software is designed as a composition of components that are reusable in nature. The architecture shall respect preservation of properties of individual building blocks once 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. More information about this can be found in this section Section 2.3.1. Also, an entire chapter Chapter 3 is dedicated for this topic. This requirement is traced to user needs:

- Shorter software development time
- Reduce recurring costs
- Increase cost-efficiency
- Support for various system-integration changes
- Product policy
- Role of software suppliers

Sofwtare observability The software architecture should provided means to observe the software specific parts and extract current and past status of the software using the services specified by its operational scenarios. This prevents the need for post launch updates or patches of the software in case of failure analysis needs. Traced to user needs:

- Quality of the product
- Reduce Verification and Validation effort
- Simplification and harmonization of FDIR
- Optimize flight maintenance

Software analysability he design process and methodology used for the reference architecture shall support the verification at design time of functional and non-functional properties. Traced to user needs:

- Quality of the product
- Reduce Verification and Validation effort

Property preservation The non-functional properties become the constraints on the system as they specify the "frame" in which the system is expected to behave and be consistent with what was predicted during the analysis. 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 properties and reactions to violation of the properties. Traced to user needs:

- Quality of the product
- Reduce Verification and Validation effort

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

- Shorter software development time
- Mitigate the impact of late requirement definition or change
- Support for various system integration strategies
- Product policy
- Role of software suppliers

Support for variability factors In order to reduce the complexity of the architecture, the potential variation of the architecture induced by the variation of the domain must be isolated in some places such as reuse is improved and need for modification is decreased. Traced to user needs:

- Increase cost-efficiency

Late incorporation of modification in the software The architecture should be immune to modification of the software late in the software life cycle. System integration 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.Traced to user needs:

- Mitigate the impact of late requirement definition or change

Provision of mechanisms for FDIR The aircraft dependability should be handled by the architecture and in particular the Fault Detection, Isolation and Recovery. Traced to user needs:

- Simplification and harmonization of FDIR

Sofwtare 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/uptime. Traced to user needs:

- Optimize flight maintenance

2.3. The Software Architectural Concept

COrDeT (Component Oriented Development Techniques) aimed at investigating various techniques in fields such as software product lines 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:

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

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 framework consisting of analysis techniques, in general, to a set of schedulability analysis equations, which help to judge using formally, whether the description of the architecture is statically analyzable.

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.

A conforming Exection Platform An execution platform helps to preserve at run-time, the properties asserted by the static analysis, and its able to react to possible violations of them.

2.3.1. Component Model

The software architectural concept is based on component based software engineering (CBSE) and the approach defines a component model that features three software entities: The Component (this is a design entity), the container and the connector (these two entities are used in implementation only and they do not appear in the design space). This approach allows creation of the OBSW as a connection of interconnected components. The execution platform defined in the software architecture provides the services to the components, container and the connectors (refer section Section 2.3.1 for more information). Then finally, all the software is deployed on the physical architecture namely the computational units, equipments, 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. 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. The Correctness by Construction practice follows:

- To give a solid reasoning on their correctness,it is necessary to use formal and precise tools and notations for the development and verification of any product item, may it be document or code
- Defining things only once so as to avoid contradictions and repetitions
- Designing the software that is easy to verify eg. 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 applied to CBSE approach based on Model-driven Engineering (MDE) approach 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 fully automated as much as possible.

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

- The components are restricted to hold the functional code only. The non-functional requirements which has effects on the run-time behavior eg. 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 which actually serve the non-functional requirements, enables the execution of the schedulability analysis directly on the model of components, makes the implementation of the non-functional concerns fully compliant with its specification.
- 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.

Inculcating separation of concerns in the development process has two major benefits:

- It increases the reuse potential of the components, which is an important high level requirement described before (refer Section 2.2.4). 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. This is also one of the high level requirements defined in the section before (refer Section 2.2.4).

Composability is guaranteed when the properties of individual components are preserved on component composition, deployment on target and execution. The components, as discussed 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. 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 (refer Section 2.3.2)chosen should help extend composability to the non-functional constraints eg. 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 model allows the execution semantics of the components with non-functional descriptors to be completely understood.

In this software reference architecture, 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 software architecture of our component-model features three distinct entities: the component, the container and the connector.

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.

The OBSW is built as an assembly of components, deployed on an execution platform which caters to the needs. 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. There is also an event based communication system 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 Section 2.3.1.

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.

Container The container is a software entity that wraps around the component, which is directly responsible for realizing of th non-functional properties. The relation between the component and the container is a famous software design pattern called the "inversion of control". 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. The container can also intercept 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". 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 (refer Section 2.3.1)as the concern 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 specify the interaction style in it's component interface.

The component can be specified independently of: The component it eventually be bound to (This is already discussed in the previous section on separation of concerns Section 2.3.1 that the same component can be deployed after annotating the component with different non-functional requirements), the cardinality of the communication and the location of the other components it connects to.

No complex connectors are necessary as the nature of the target systems reduces the variety of connectors needed. Connectors necessary for function/procedure calls (which ar usually straight-forward), remote message passing or data access (I/O operations on files in the memory) are sufficient in most cases.

2.3.2. Computational model

Using a computational model is required by the Space Software engineering standard (ECSS-E-ST-440C). A dynamic software architecture is described according to an analyzable computational model, i.e from the description a schedulability analysis can be conducted. Computational model is concerned about entities that belong to the implementation model (eg. tasks, protected objects and semaphores). A more abstract level description of those entities should be provided so that:

- Pollution of the user-models with entities that are more primitive and is of interest to the lower levels of abstraction, is avoided.
- They represent those entities and their semantics faithfully.
- Ensures correct transformation of the information set by the designer in the higher-level representation to entities recognized by the computational model.

2.3.3. Execution platform

The execution platform is a part of the software architecture proving all th necessary means for the implementation of a component and computational model. It is a 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 of services:

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 2.3.1)

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.

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 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 or the development of part of the software. 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 and 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 leads to a iterative/incremental development process. Component detailed design and implementation is usually done by the software developers are contracted 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 when a particular computation model and execution platform are adopted (The choices made for this Master thesis are explained in the following chapters)

The following entities belong to the design space: Data types, events, interfaces, component types, component implementations, component instances, component bindings and the entities 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:

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

Step 3: Definition of component types Component types form the basis of reusable software asset. 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 consist of:

- One or more provided interfaces, which list the services that the component of this type would provide
- 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. They are local to the component and they 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 implementations and all of these implementations contain only pure sequential code and 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. This helps in determining the communication budget allocated because the size of data types used for communication is already known. Component implementation is thence a particularly attractive unit to be subcontracted to third-party because the software architect can define components, attached 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.

Step 5: Definition of component instances A component instance is an instance of a component implementation. It is the deployment unit subject to allocation on a processing unit and it is on which the non-functional properties are specified. Specifically, they are attached to the provided interface side of the component, as they are specifically the expression of a property or a provision of the component instance. There will not be any non-functional concerns attached to the required interface side of the component.

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 and 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 dispatch to the correct operation in the bound provided interface. The signature of the call calling operation in the RI (required interface) and the called operation in the PI (provided interface) are different and the connector is in charge of performing this step and 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 trace bindings between an event emitter port of one component and an event receiver port of another component.

Step 7: Specifiction 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. 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, the operation is executed by a dedicated flow of control in the callee.

An `immediate` operation is said to be `protected` if it needs to be protected from data races in case of concurrent calls and it is said to be `unprotected` if it is free from such risks. In case of a `deferred` operation type, the architect must chose one of the following release patterns of 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 nter-arrival time (MIAT). The execution platform and the infrastructural code guarantees this MIAT separation between two subsequent calls to the operation and the component implementor 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 implementor does not have to worry about it as for the sporadic operation.

For all the operations which have concurrency as `deferred`, the softare 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.

The Component model of the OSRA also provides the software architect an option to define measurement units (conversion factors between them) and reuse the definitions across other projects. Non-functional attributes eg. Period of a deferred operation has a value and a measurement unit with it.

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. 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 o 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 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 requires a model representation of the containers and connectors, and as the topic of this master thesis is to generate containers and connectors, it is not possible to do an accurate model based schedulability analysis at the current phase of development. More about model based schedulability analysis done in another project ASSERT is available in the Appendix section.

Step 11: Generation of containers and connectors This step is one of the main focus points of the master thesis. 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 (which indicates that this component is for interaction purpose only) components 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 meet all its user needs and high level requirements.

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 te commandability and observability of the spacecraft are specified

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 generated infrastructure code with the model-code transformations through the help of a simple example:

5.2.1. Problem description

Two components, namely `ComponentA` and `ComponentB` are constructed.

`ComponentA` has a requirement that the function that it implements `startOperation` needs to be called on the periodic basis with a period of two seconds. `ComponentA` also has two required interface ports. One required interface port allows `ComponentA` to call an operation named `operationAdd` in `ComponentB` with concurrency kind `immediate` and the other required interface port allows `ComponentA` to call the same operation `operationAdd` with concurrency kind as `deffered`.

`ComponentB` has a requirement that it must implement the operation `operationAdd` which can be called by any other component connected to it. As the component model does not restrict on the number of component implementations which can be part of a component, `ComponentB` implements two versions of the same operation `operationAdd`. `ComponentB` has 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. 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.

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.

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

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

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

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.

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 43 erneut die komplette Choreographie.

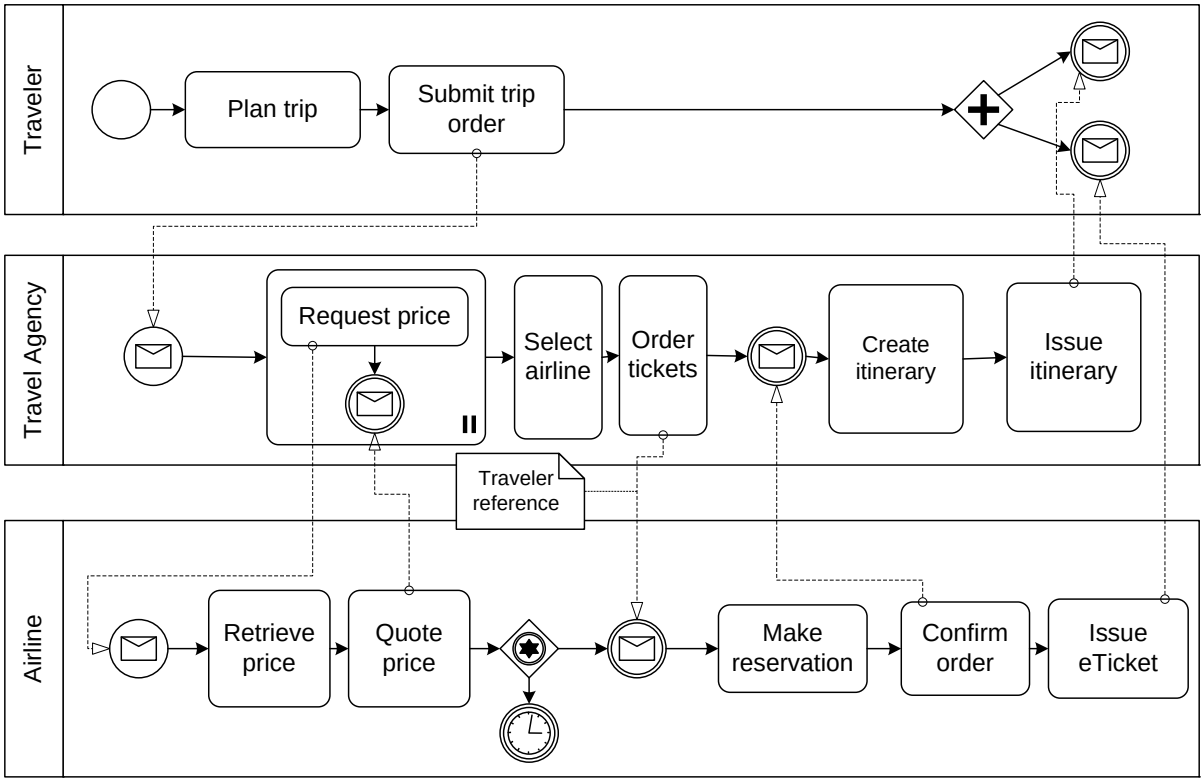


Figure A.1.: Beispiel-Choreographie

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.

A.7. Pseudocode

Algorithm A.1 zeigt einen Beispielalgorithmus.

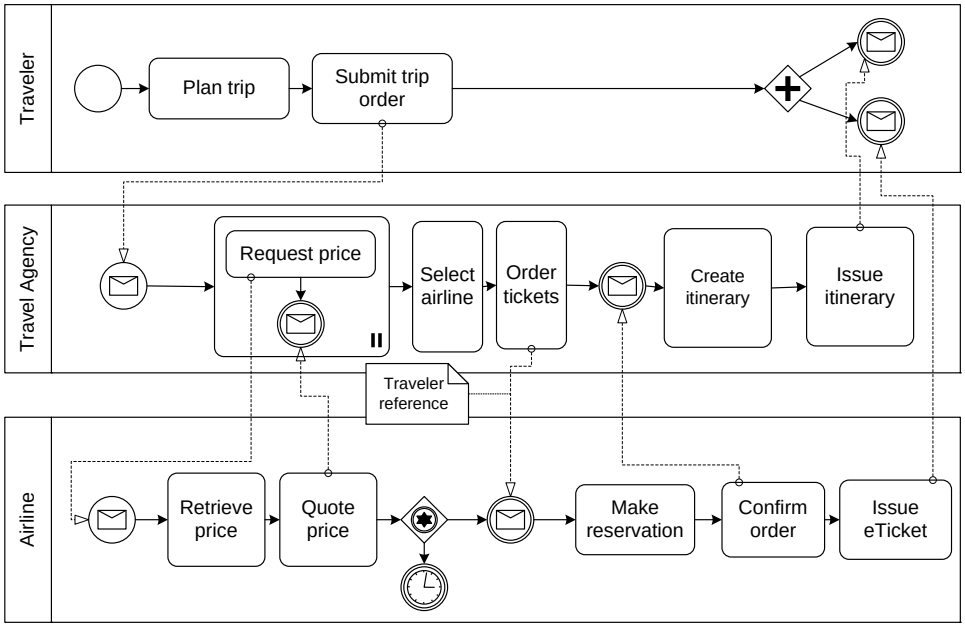
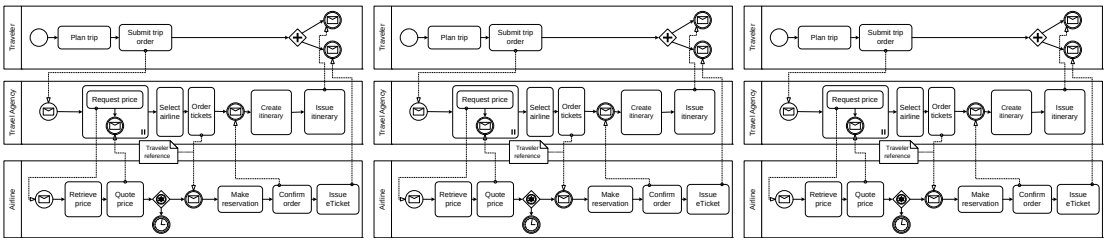


Figure A.2.: Die Beispiel-Choreographie. Nun etwas kleiner, damit `\textwidth` demonstriert wird. Und auch die Verwendung von alternativen Bildunterschriften für das Verzeichnis der Abbildungen. Letzteres ist allerdings nur Bedingt zu empfehlen, denn wer liest schon so viel Text unter einem Bild? Oder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

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 36". 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 35", `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.

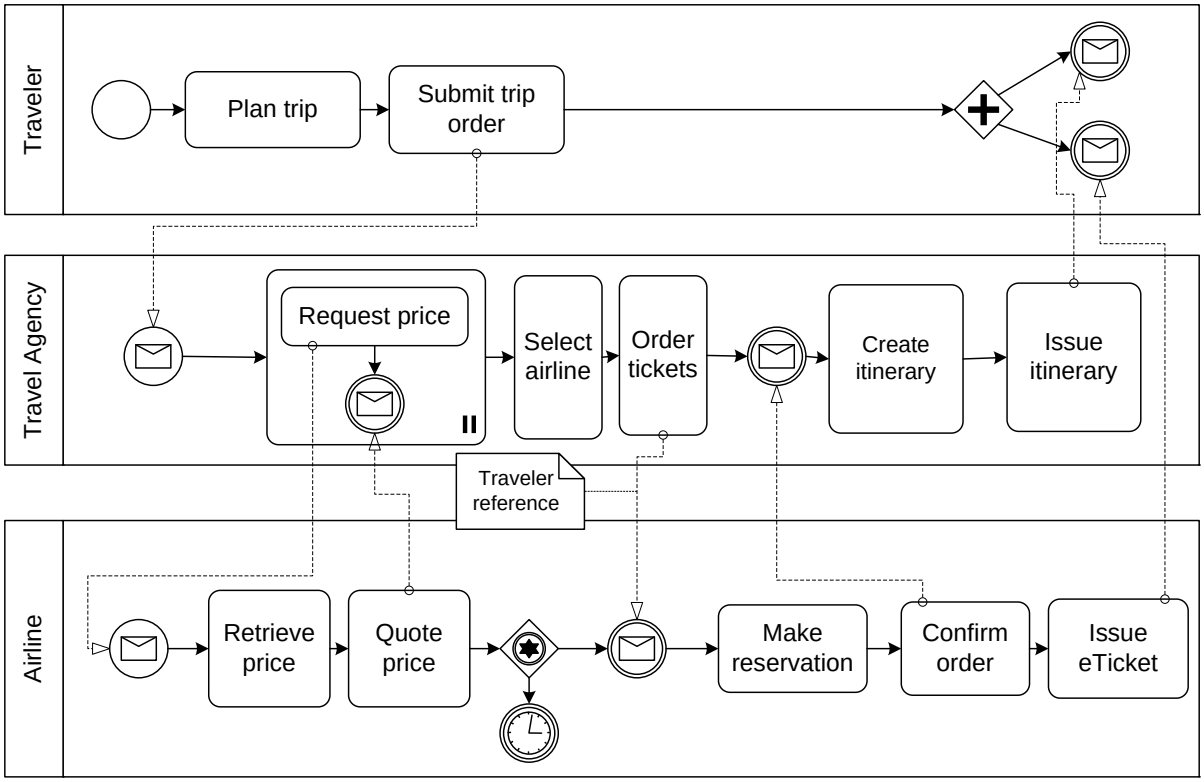


Figure A.4.: Beispiel-Choreographie I

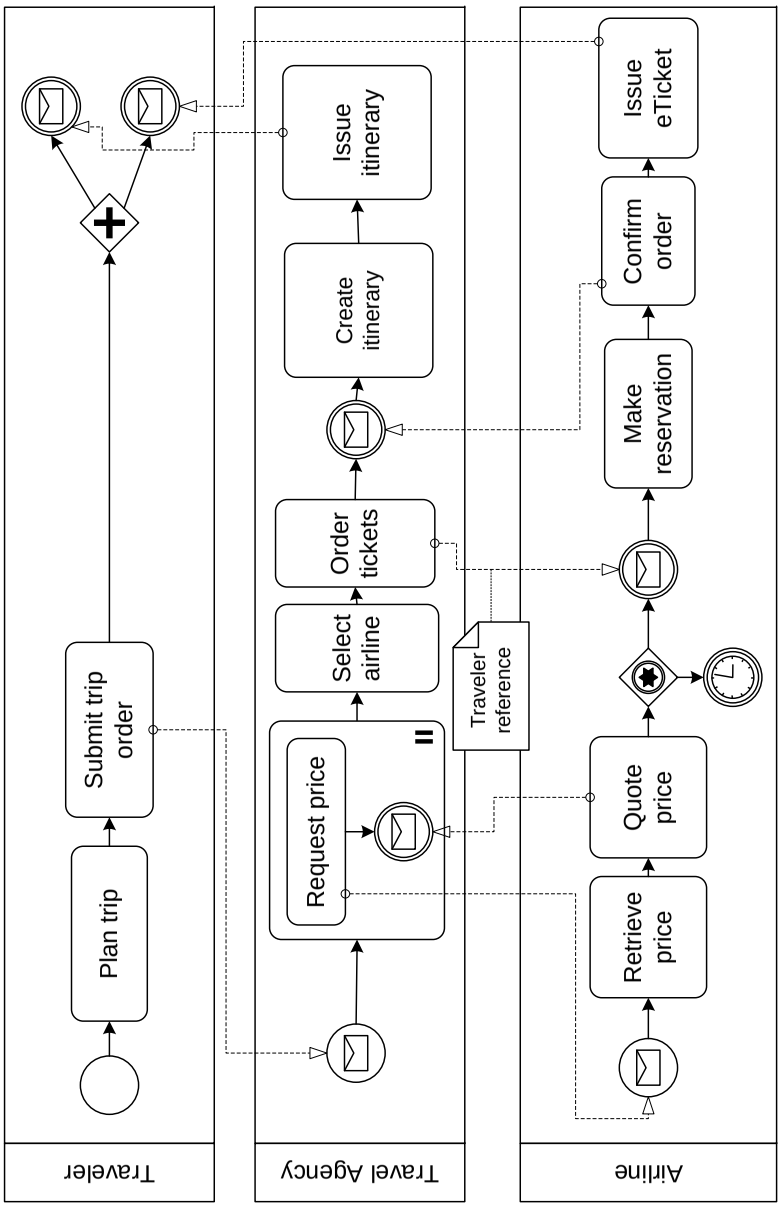


Figure A.5.: Beispiel-Choreographie II

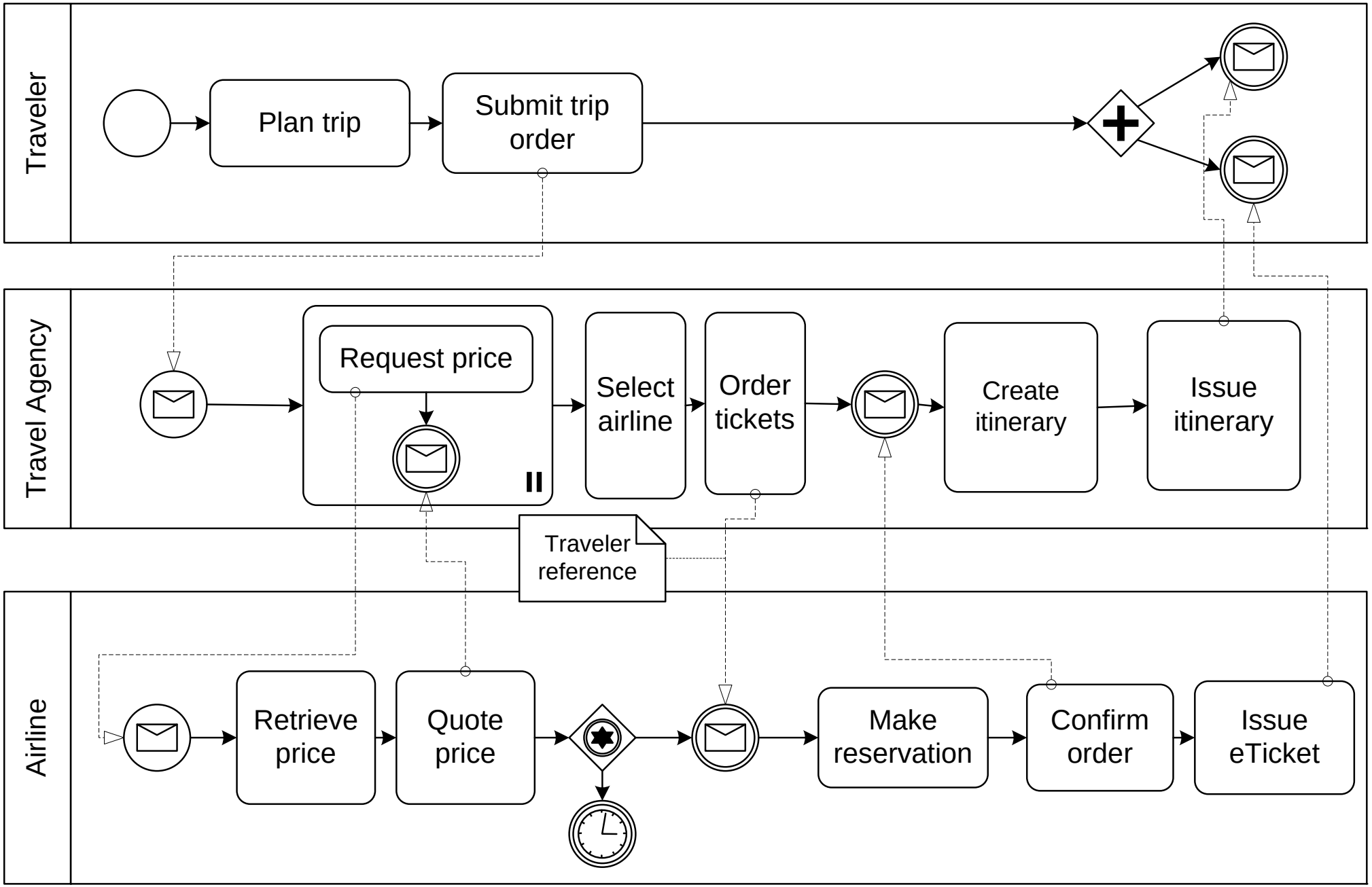


Figure A.6.: Beispiel-Choreographie, auf einer weißen Seite gezeigt wird und über die definierten Seitenränder herausragt

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