UNIVERSITY OF OULU

DEGREE PROGRAMME IN ELECTRICAL ENGINEERING

MASTER'S THESIS

Service-Oriented Approach in Base Transceiver Station Platform Software Design Utilizing Domain-Specific Modeling

Author	Veikka Grekula
Supervisor	Mika Ylianttila
Second Examiner	Timo Bräysy
Technical Advisor	Tiina Rantala

February 2018

Grekula V. (2018). Service-Oriented Approach in Base Transceiver Station Platform Software Design Utilizing Domain-Specific Modeling. University of Oulu, Degree Programme in Electrical Engineering. Master's Thesis, 80 p.

ABSTRACT

Software (SW) systems are becoming more and more complex due to the need of functionalities in the systems. In the component-based SW design approach, the software is modeled as software components and these components are becoming too extensive to be able to model a SW feature efficiently. A service-based approach can offer an alternative for solving the rising problem of an inefficiency among the component-based SW modeling. Instead of being responsible for the different SW components, in the service-based approach, the SW functionalities are divided into one or more services and further into micro-services.

In this thesis, the possibility of replacing the component-based software modeling approach with the service-based SW modeling approach is studied. In this work, an existing set of SW features, that are modeled with a component-based approach, is modeled using the service-based approach. Model-driven software development (MDSD) methods, such as Model Driven Architecture (MDA) and domain-specific modeling (DSM), are utilized to create a service-based solution. The aim of this thesis was to implement a functional service-based model from which the reports can be generated as an output.

The created domain-specific modeling language (DSML) and the different abstraction layers of the created model are described in detail. The modeling language and the proposed metamodel were created using MetaEdit+ metamodeling tool provided by MetaCase. The code generators were implemented using MetaEdit+ reporting language (MERL) which is an object-based scripting language. The created service-oriented architecture and the modeling language were evaluated based on the theory, user experience and the reviews of the SW specialists.

The evaluation of the proposed metamodel, modeling language and the service-oriented architecture (SOA) stated that the created modeling language and the service-based approach for the SW modeling fulfils the requirements of the DSML and SOA. However, some questions emerged concerning the size of the service and the possibility to create functional entities simultaneously in a faster and efficient way. Due to the promising results of this thesis, future work could investigate the suitable size of a service that the component-based approach can be replaced by the service-based approach by means of efficiency.

Key words: Model-driven software development, code generation, metamodeling.

Grekula V. (2018) Palvelupohjainen lähestymistapa tukiasema-alustan ohjelmistosuunnittelussa hyödyntäen aluekohtaista mallinnusta. Oulun yliopisto, sähkötekniikan tutkinto-ohjelma. Diplomityö, 80 s.

TIIVISTELMÄ

Erilaisten toiminnallisuuksien ja ominaisuuksien kasvanut tarve tekee ohjelmistoista yhä vaativampia toteuttaa. Komponenttipohjaisessa lähestymistavassa ohjelmistosuunnittelussa ohjelmistot on mallinnettu komponentteina. Näiden komponenttien sisältö on tullut liian laajaksi, jotta niitä voidaan hyödyntää tehokkaasti erilaisten ohjelmistotoiminnallisuuksien mallintamiseen. Myös eri komponenttien yhtäaikaisesta hallinnasta on tullut haasteellista komponenttien rakenteen vuoksi. Palvelupohjainen lähestymistapa voi tarjota ratkaisun komponenttipohjaisen lähestymistavan tehottomuuteen toiminnallisuuksien mallintamisessa. Palvelupohjaisessa lähestymistavassa ohjelmistotoiminnallisuudet on jaettu eri palveluihin, joista jokainen on vastuussa yhdestä laajemmasta osa-alueesta.

Tässä diplomityössä tutkitaan mahdollisuutta korvata nykyinen komponenttipohjainen ohjelmistosuunnittelun lähestymistapa palvelupohjaisella lähestymistavalla. Työssä mallinnetaan olemassa oleva komponenttikohtaisella lähestymistavalla mallinnettu toiminnallisuusjoukko palvelukohtaisella lähestymistavalla. MDSD-menetelmiä (Model-Driven Software Development), kuten MDA (Model Driven Architecture) sekä DSM (Domain-Specific Modeling), on hyödynnetty luomaan palvelupohjainen ratkaisu. Diplomityön tavoite on toteuttaa toimiva palvelupohjainen malli käyttäen aluekohtaista mallinnuskieltä, josta koodigeneroinnin avulla voidaan generoida raportteja.

Luotu aluekohtainen mallinnuskieli ja luodun metamallin eritasoiset abstraktiokerrokset on kuvattu yksityiskohtaisesti. Metamalli ja mallinnuskieli on kehitetty käyttäen MetaCase:n tarjoamaa MetaEdit+-metamallinnustyökalua. Koodigeneraattorit on luotu olio-ohjelmointiin perustuvalla MERL-ohjelmointikielellä (MetaEdit+ Reporting Language). Luodun palvelupohjaisen arkkitehtuurin ja mallinnuskielen arviointi perustui niiden taustalla olevaan teoriaan, käyttäjäkokemukseen sekä ohjelmistoasiantuntijoiden katselmointikommentteihin.

Luodun metamallin ja mallinnuskielen arviointi osoitti, että molemmat täyttivät hyvin niille asetetut vaatimukset. Arviointi herätti myös kysymyksiä koskien palveluiden kokoa ja mahdollisuutta luoda yhtäaikaisesti useita toiminnallisia ohjelmistokokonaisuuksia nopeammin ja tehokkaammin. Lupaavien tulosten myötä jatkossa voitaisiin tutkia palveluiden laajuutta ja mahdollisuutta löytää optimikoko palveluille, jotta nykyinen komponenttipohjainen lähestymistapa voitaisiin korvata palvelupohjaisella lähestymistavalla, jolloin mallinnuksesta tulisi tehokkaampaa.

Avainsanat: MDSD-menetelmä, koodigenerointi, metamallinnus.

TABLE OF CONTENTS

FOREWORD

This Master's thesis was done for Nokia in Oulu from September 2017 to February 2018. In addition, this thesis was part of the ECSEL JU funded MegaM@Rt2 project. The aim of this thesis was to develop a new service-oriented architecture that could replace the current component-based structure in the modeling environment by utilizing domain-specific modeling languages. Moreover, this thesis was done in a software development team. The work was implemented by using MetaEdit+ software. The consulting from MetaCase during the writing of this thesis was indispensable. Especially, I sincerely thank the CEO of MetaCase, Dr. Juha-Pekka Tolvanen, for the advice and support during the writing of the thesis. His contribution to this thesis and knowledge of metamodeling was priceless.

I would like to thank my supervisor, prof. Mika Ylianttila, and second examiner, Timo Bräysy, from the Centre for Wireless Communications at the University of Oulu. Their contribution helped me to finalize this thesis.

The biggest thanks go to Sami Ansamaa and Orvo Mikkola for offering me the opportunity to work at Nokia. Their organizing skills were valuable to this thesis. I want to thank my team mates at Nokia for supporting me. Especially, I want to thank Janne Mäkäräinen for giving me technical support during the work. In addition, I would like to thank my technical advisor Tiina Rantala. Her contribution through the work was irreplaceable.

At last, I want to thank my family, friends and fellow students for their support. The biggest thanks go to my girlfriend for providing me with not only support but also the proofreading of this thesis.

Oulu, 21.2.2018

Veikka Grekula

LIST OF ABBREVIATIONS AND SYMBOLS

3GPP	3rd generation partnership project
API	application programming interface
BB	baseband block
BIP	BTS intranet protocol
BTS	base transceiver station
BVA	boundary value analysis
CASE	computer-aided software engineering
CBSE	component-based software engineering
CCB	control and clock block
CIM	computational independent model
DSL	domain-specific language
DSM	domain-specific modeling
DSML	domain-specific modeling language
ECP	equivalence class portioning
EM	event machine
ETSI	European Telecommunication Standards Institute
FDD	feature-driven development
GOPPRR	graph object property port relationship role
GPML	general purpose modeling language
HTML	hypertext markup language
HW	hardware
HWAPI	hardware application programming interface
ICOM	internal communication
IDE	integrated development environment
IF	interface
IT	information technology
LTE	long term evolution
MBT	model-based testing
MDA	Model Driven Architecture®
MDD	model-driven development
MDSD	model-driven software development
MERL	MetaEdit+ reporting language
MWB	MetaEdit+ workbench
OBSAI	open base station architecture initiative
OMG	Object Management Group®
OO	object-oriented
PIM	platform independent model
PSM	platform specific model
QoS	quality of service
RFB	radio frequency block
RNC	radio network controller

RP	radio platform
RP	reference point
SC	system component
SDL	specification and description language
SE	system element
SOA	service-oriented architecture
SQL	structured query language
sRIO	serial rapid input/output
SS	subsystem
SUT	system under test
SW	software
SysCom	system internal communication
TB	transport block
TDD	test-driven development
UML	Unified Modeling Language®
XML	extensible markup language
XP	extreme programming

1. INTRODUCTION

Building a large-scale software (SW) system is a complex task. The increased need for software and functionalities behind the software has made the system design work more and more demanding. The aim of SW modeling is to model those functionalities as efficiently as possible retaining the reusability of the SW models. When it comes to modeling, the SW techniques are constantly developing. Still, there is room for a technique to model software by means of efficiency. The currently used component-based approach in SW modeling is problematic due to the construct of the large-scale enterprise. The large amount of SW functionalities is divided into SW components, and these components are becoming too large to be able to work efficiently. Service-oriented architecture (SOA) is one solution to build an enterprise-scale software product from services. SOA consists of services that communicate with each other using well-defined interfaces. Hence, SOA offers an interface-based approach that concentrates not only on reusable services, but also on defining interfaces as efficiently as possible. That is, SOA defines the services of which the software system is composed and describes the interfaces and interactions among the services to realize a specific behavior. [1]

In this thesis, the possibility to create an alternative for component-based software design using service-based approach in base transceiver station (BTS) software development process is researched. One solution to enable SOA is utilizing domain-specific modeling (DSM) and its domain-specific modeling language (DSML). DSM is used to model an existing feature of BTS software in internal communication (ICOM) area, which includes, for example, Serial Rapid Input/Output (sRIO), system internal communication (SysCom) and Ethernet functionalities. Also, DSM utilizes model-driven software development (MDSD) approaches such as metamodeling and agile methods.

DSM is a MDSD approach that has two main targets. First, DSM defines special-purpose languages to achieve domain concepts. Second, DSM enables the use of code generators to get a valid code from a model. This thesis concentrates on the first approach, but also the code generators are investigated and implemented. The tool support enables DSM. To enable DSM solution, MetaEdit+ tool is used as a DSM tool in this work. The challenge for SOA is to integrate different models and abstractions. DSM raises the abstraction level. Thus, DSM is a potential approach to realize SOA. In this work, the whole modeling process of a SW feature from start to end, including the theoretical and concrete aspects of the modeling process, is described in detail. The aim is to implement SOA using MDSD approaches. Models are designed using MetaEdit+ tool. In addition to creating DSM solution, it is also evaluated via user experience and via reviews by SW specialists. MetaEdit+ tool offers also a code generation option that is utilized to demonstrate the correctness of the proposed SOA. However, in this work, the service-oriented modeling approach is the priority. [2]

This thesis is structured as follows. In chapter 2, basic information of the BTS system platform SW is provided. In addition, the phases of software technology evolution are introduced and software testing strategies are described. Moreover, a brief introduction to model-based testing and software development methods is given. Chapter 3 focuses on MDSD and its approaches. Metalevels, levels of abstraction and the difference between diagrams and models are described. Furthermore, DSML development processes and guidelines as well as the metamodeling tool MetaEdit+ are introduced. Chapter 4 contains the actual work: how the SOA, metamodeling and DSML are used to design the SW features. In chapter 5, the results are gathered and discussed. Also, a brief evaluation of the metamodeling tool MetaEdit+ is given. Finally, chapter 6 provides the summary and concludes the work.

2. SOFTWARE SYSTEM DEVELOPMENT AND TESTING

This chapter concentrates on the software system development. First, the basic information of BTS system platform SW is introduced. Second, an overview of the software technology evolution is described, including object-oriented, component based and service-oriented software. Finally, a brief introduction to the model based testing and different software development methods is given.

2.1. Basic Information of the BTS System Platform SW

Base transceiver station (BTS) platform SW is a software system that is discussed in this work. The main responsibility of the BTS platform SW is to hide the hardware environment for the SW applications. Figure 1 presents the SW modularization concept. Network element (NE), e.g. LTE BTS, is a combination of SW and HW and it consists of different system components (SC) and system elements (SE). SE cannot be decomposed and it is an independent SW entity. In addition, SCs are software that can exist independently or further consisting of services. Also, services can be independent or consist of subsystems (SS). To be able to communicate internally and externally, every SC provides internal and external interfaces. Moreover, If the SC wants to communicate with the HW, they need to use interfaces that are provided by HW components. Table 1 summarizes the SW modularization concept entities.

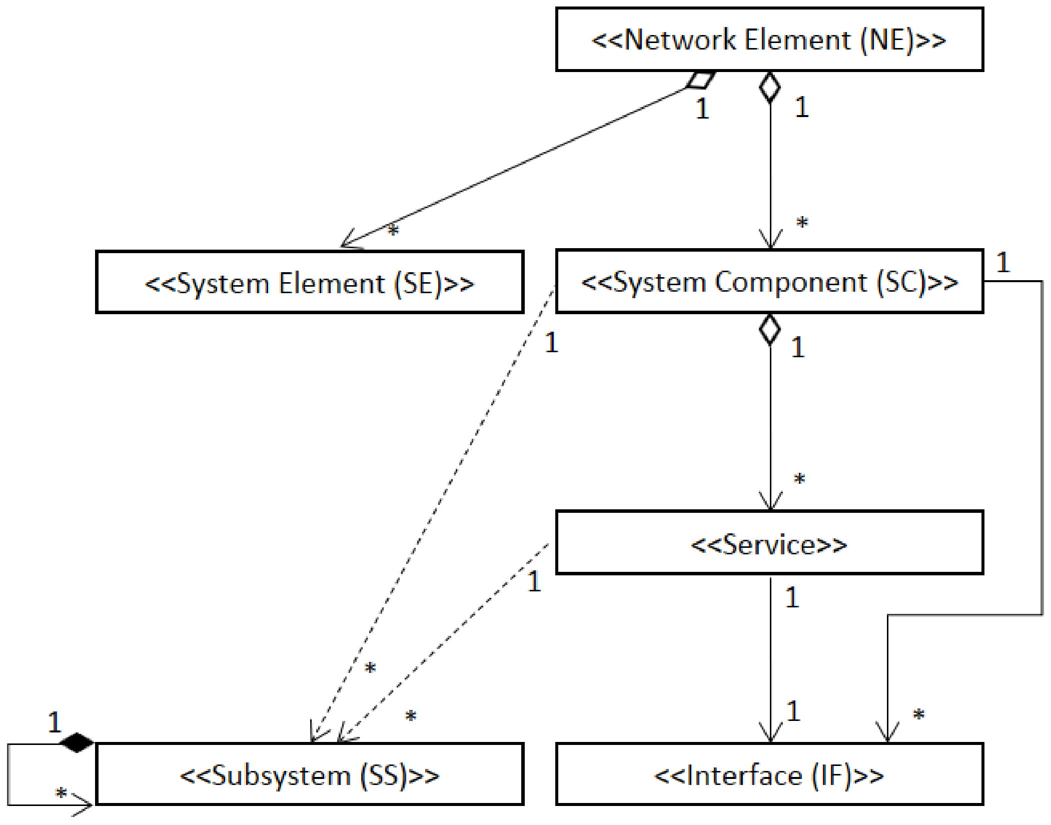


Figure 1. BTS SW architecture modularization.

Table 1. Definition of SW modularization concept entities

Entity	Description
Network Element (NE)	NE is a network entity in the network that provides a set of functionalities.
System Element (SE)	SE is part of the BTS SW, but cannot be decomposed. The BTS SW defines responsibilities for the SE, but it is only seen through the services and its interfaces.
System Component (SC)	SC encapsulates some functionality area of the NE. A SC is a collection of services and SSs and can be further decomposed into further entities.
Service	Service is an implementation of a functionality that is delimited by a one interface. Service may have its implementation modeled via SS.
Subsystem (SS)	SS s a realization method of one or more services. SS can include other SSs.
Interface (IF)	IF is a defined entry point of a functionality provided by a SE, SC or service.

BTS consists of radio frequency (RF) modules and of a system module (SM). Open Base Station Architecture Initiative (OBSAI) has defined a complete reference architecture for BTS [3]. The following elements are the main elements of the architecture [3]:

- Functional blocks consisting of the baseband block (BB), RF block (RFB), control and clock block (CCB) and transport block (TB).
- External network interface, e.g. Iub to the radio network controller (RNC) for 3rd generation partnership project (3GPP) systems.
- Internal Interfaces between BTS functional blocks, such as reference points (RP) 1, 2 and 3.
- External radio interface, e.g. Uu to the user equipment (UE) for 3GPP systems.

The BTS platform SW is part of the SM. Figure 2 shows that SM includes all the functional blocks except for RF Blocks [3]. The Transport block consists of at least one module that performs functions such as external networks interface, internal networking, quality of service (QoS), synchronization and security functions. The control and clock block is the primary control processor for the BTS and it consists of at least one module. The BTS status and resources are controlled by the CBB. The baseband block also consists of at least one module that executes baseband processing for the air interfaces. [3]

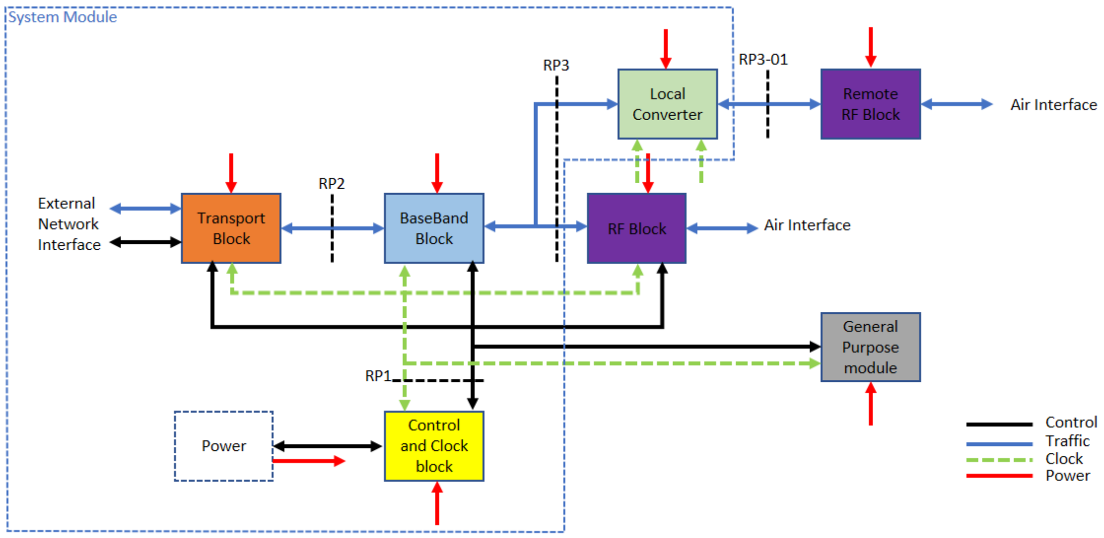


Figure 2. BTS reference architecture.

The hardware environment is controlled and regulated by using the BTS platform SW. Moreover, the BTS platform SW provides different services for application through specific platform Application Programming Interfaces (APIs). These APIs are used to provide a communication link between the BTS platform the SW and SW application which can be, for example, radio access technology (RAT) SW. Further, RAT can be, for example, long term evolution (LTE). By using the communication link, the BTS platform and the SW application can request different services. The requests are handled in the SW side of the platform, and decisions of which services at that point are used are done based on the existing knowledge of the present status of the SW system. To be able to control the status of the system HW, the BTS platform SW has multiple specific interfaces. Figure 3 shows an example of the interfaces, and how they are linked in the radio platform SW (RPSW). In this figure, interfaces from IF1 to IFn are considered as external interfaces. The RPSW is considered to be a black box in such a way that it hides the used SCs. In other words, the client does not see the SCs inside the RPSW, it only sees the interfaces. In this case, the links to the SCs of the RPSW are presented as ports. [4]

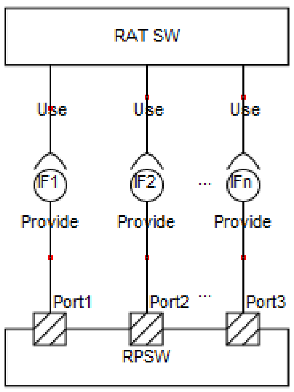


Figure 3. Example of RPSW external interfaces to RAT SW.

Figure 4 shows an example of some of the internal interfaces between the SCs of the RPSW. There are the provided and the used internal interfaces between the different SCs. In the example, there are three different SCs, SC1, SC2 and SC3, that can either provide and/or use an interface. Interfaces SC1 IF1 and SC3 IF2 are external interfaces, and interfaces SC2 IF1 and SC3 IF1 are internal interfaces. For example, SC1 provides an interface SC1 IF1 that is used by RAT SW and uses an interface SC2 IF1 that is provided by SC2. This thesis concentrates on the SW that is part of the internal communication. [4]

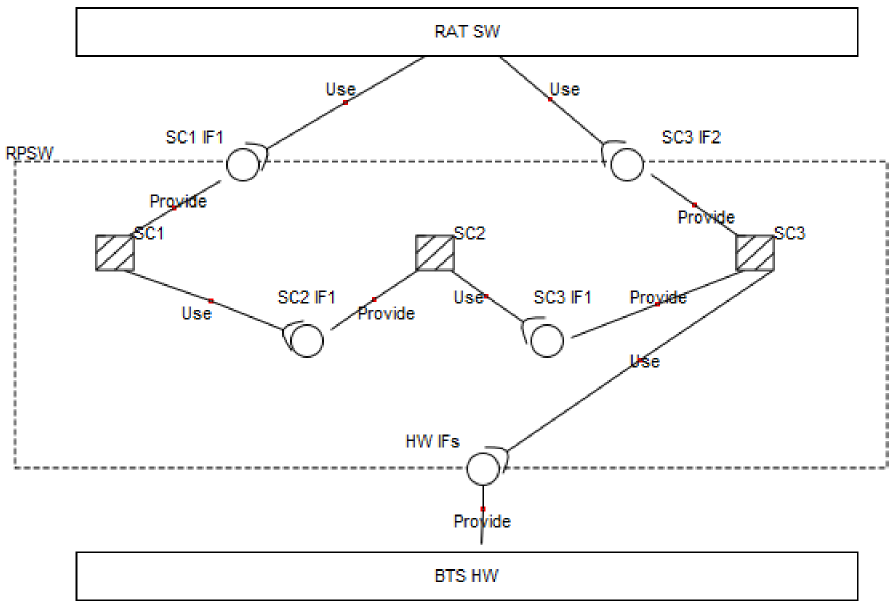


Figure 4. Example of RPSW external and internal interfaces.

2.2. Software Technology Evolution

Like every technology, a software technology and programming languages are constantly developing. New tools and technologies are needed to fulfil the requirements and needs of the new software designs and software developments. Therefore, innovative approaches to software design and development in the information technology (IT) industry are continuously searched. As a result, software development and programming languages have experienced an extreme evolution. [5]

The history of the software technology evolution can be divided into three main parts. First, in the beginning of the 1990s, the concept of object-oriented (OO) languages arose to depict the concrete problems. OO languages allow to write reasonably easy code to relate to concrete problems. Next, the demand for automation of the complete business process was discussed. To help the automation process, component-based programming was introduced. Component-based software engineering (CBSE) arose in the late nineties providing advantages, such as increased management of complex problems, reduced development and increased productivity. The limitations of OO development to support component reuse was one of the main reasons to develop CBSE. With component-based programming it was possible to automatize the whole business process. Finally, the third approach, service-oriented programming, was introduced. Web-based environment and the increased demand for software were the basis for developing SOA. In Figure 5 [5], the different phases of the software technology evolution are gathered under the technology evolution box. Each of the following technologies are introduced one by one in the next sub-chapters. The line between the different phases of the SW technology evolution is indistinct, therefore often there is no clear evolution step that is being worked with. Therefore, in the SW design and development, there might be situations where parts of each evolution step are used at the same time. [5]

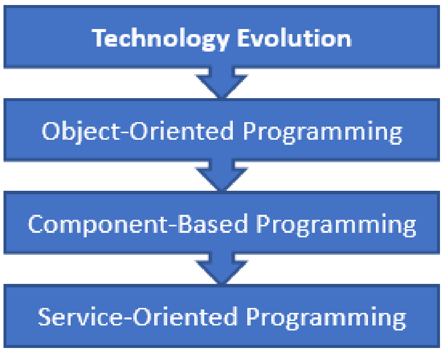

Technology Evolution
Object-Oriented Programming
Component-Based Programming
Service-Oriented Programming

Figure 5. Software technology evolution.

2.2.1. Object-Oriented Software

The aim of the OO system is to reduce the complexity of the software by using abstractions. An abstraction is a concept that makes it easier for the software engineer to deal with details. There are two main types of abstractions that are combined in object-oriented systems: procedural abstraction and data abstraction. [6]

Procedures, also known as known as functions or routines, is the basis on which software relies. These procedures enable procedural abstraction. When one procedure is used, the programmer does not need to care about all the details of how the computations are performed. The programmer only needs to know how to call the procedure, and what the result of the computation is. This is known as procedural abstraction. Procedural abstraction works when the aim is to work with a simple data. Nowadays, programs and applications are more and more complex. Therefore, an engineer must work with multiple different data, and the system written by using procedural abstraction can be very complex. [6], [7]

The other abstraction, data abstraction, is a helpful concept when the complexity of the system needs to be reduced. The main idea is to gather all the pieces of the data that are somewhat similar so that the data can be seen as a unit that is easy to modify as a whole. When a software application was described only by using either procedure or data abstractions, it was seen that the whole procedure is way too simplistic. As a solution, the concept of object-oriented programming was presented. [6], [7]

In OO programming, the software is not divided into data or procedures anymore, but rather into objects; so called abstract software artifacts. Basically, object-oriented artifacts consist of classes and their instances, which are called objects. In an object-oriented program, classes are entities of data abstraction and they represent a set of similar objects. In other words, objects that share the same behavior and properties are instances of a one class. Usually, a class contains at least a code that describes the structure of the objects of the class and methods that are procedures to execute the behavior of the objects. In general, if something could have instances, it should be considered as a class, and if something is distinctly a member of the set described by a class, it should be considered as an instance. [6], [7]

As mentioned, objects are instances of a class and they have certain properties. Classes and objects are tied together and basically cannot be discussed independently. The values of the properties specify the objects by describing the current state of the object. The behavior tells how the object acts and reacts when the state changes. The objects depict all the essential things that are fundamental to the users of the program. A variable in OO is the place where the data is put. Each class notifies a group of variables corresponding to the data that belongs to each instance. In OO program, it is important to realize the differences between variables and objects. Variables can refer to a specific object or to no object at all. When the variable refers to an object, it is known as a reference. Variables can refer to multiple objects at the same time. The type of the variables defines which kind of objects it could contain. [6]

The next important thing when talking about OO is the concept of instance variables. Instance variables can be divided into two groups depending on the target of the implementation. They can be used to implement attributes or to implement associations. Attribute in this context is a piece of data that is used to depict the properties of an object, i.e. a name, whereas, an association depicts the relationships between instances of classes. [6]

Encapsulation is an essential concept in the field of OO software systems. A class behaves as a container to hold its variables and methods. It also assures that the object can be handled independently from rest of the software system. In other words, if there are changes in the software systems, the object keeps its integrity and functionality. Encapsulation offers simplicity and clarity in such a way that there is no function or data in the program that is not included into any object. Encapsulation is highly linked to the information hiding. [6]

Information hiding is achieved by encapsulation. It is a concept that hides the data which might be affected during the implementing process. The data is insulated from the direct access by outside objects. The key factor of information hiding is to decide whether the information is visible or hidden [8]. Information hiding brings up the concept of access levels. The attributes and methods of a class can be presented as a public, protected or private access [9]. The process where objects of one class obtain the properties of objects of another class is called inheritance. The idea of reusability is provided by inheritance. Thus, additional features can be added to an existing class without modifying it. This method is achievable by creating new classes from the existing ones. As a result, new classes have combined features of both classes. The inheritance mechanism allows to reuse classes without causing any unwanted side effects. [9]

The final bases of OO are methods, operations and polymorphism. Methods are procedural abstractions and the behavior of a class is implemented by using methods. An operation is an even higher-lever procedural abstraction from the methods that is used independently from of any code that is connected to that behavior to depict a type of behavior [6]. Polymorphism is a concept whose aim is to separate classes and their instances to be accessed in the same way. Thus, by definition, it can be said that a single object can appear in multiple forms. Therefore, under different circumstances, an object can behave differently even if the given message is the same. [7]

To summarize, OO is strongly related to the concept of modularity. This concept consists of various independent components which are implemented to function together. Modularity is a concept which increases the reusability, workability and efficiency of the software components by partitioning programs into smaller modules, while at the same time reducing complexity. [7]

2.2.2. Component-Based Software

The main idea of CBSE is to design and develop software by systems using reusable components. These components in the SW environment are quite abstract and capable of achieving a specific functionality. The component is selected based on its characteristic, such as reusability, and then assembled with a well-defined SW architecture. The concept of reuse is in a central role when CBSE is discussed. Components are created in such a way that they can be reused in other similar applications. The aim is that the system consists mainly of components. [3]

A component is a software object that is made to interact with other components by sealing a certain set of functionalities. It is important that a component has a well-defined interface in order to be able to communicate with other components. Therefore, interfaces are the most essential part in the component-based structure. All the services and functionality of the component are provided through its interface. The interfaces include services and describe the interaction of the client and the component. At the same time, the underlying details are hidden. Based on the predefined schema, the interfaces are specified. [10] The component-based approach does not build systems from a scratch; therefore, the reuse of components is the key factor [3]. Thus, the focus from a new system development is shifted to the integration of existing components to perform new tasks [10]. Moreover, in the visual language based system development environment, particularly in modeling environments, components can be further divided into metamodel, model and code components. Modeling and metamodeling are discussed in more detail in chapter 3. Commonly, components are standardized, independent, composable, deployable and documented. [3], [10]

As previously said, the concept of reusability is in a central role in the component-based software. There are several techniques for reuse. In this context, white box, gray box and black box reuse is discussed. For example, if the exact required code component can be found, the black box technique is adapted during the reuse procedure. Otherwise, the gray or white box techniques are applied. On the contrast, in the modeling environment, on which this thesis focuses, model component and metamodel component are more interested in the process, transitions and rules are defined inside the components. Hence, the component content needs to be visible during the reuse process. Therefore, the main approach for model component and metamodel components is the white box testing in the means of reuse. [10]

2.2.3. Service-Oriented Software

Service-oriented architecture (SOA) is a collection of services that communicate with each other. Each of these services has a certain collection of well-defined functions that are provided for other services via interfaces. Interfaces are typically expressed as messages and functions including their limitations. Further in this work, the word "operation" includes both messages and functions. In other words, operations are defined by an interface. These operations move between a service and a client. These movements of operations between services and client follow some set of patterns, of which the most commonly used pattern is known as the request and reply pattern. With this pattern the client sends a request message to service and the service responds with a reply message that is retuned to client. In addition, the request and reply pattern works also between services. Operations are used to provide services to end-user applications and other services in the SOA. Together, the services implement the entire system by interacting with each other. [1] According to [1], service is defined as "… generally implemented as a course-grained, discoverable software entity that exists a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model." From a SOA point of view, a service is some set of an application functionality.

The SOA is not an entity that can just be collected as if it was a grocery item on a shelf. There is no clear line whether the SOA is defined just to a specific technology or a product. The SOA is a bigger aspect. Further, the SOA is more than just services; it includes three kinds of participants and their relationships. These participants are the service provider, the service registry and the service requestor, also known as a client. A service provider and a client are software entities. The main task of a service provider is to implement service specifications. The client calls the service provider through an interface. The service registry is a repository. Figure 6 shows the relationships of those three participants [3]. The relationships between those three participants involve the publish, find and bind relations. The relations act upon the service artifacts, the service description and the service implementation [3]. All the constraints and policies of the service are specified by the service description. Usually, the service description of a service is defined by the service provider which publishes it to the service registry. The service description defines the information, such as interfaces and functionalities, that is needed in order to use a service [11]. The client uses a find relation to get the service description from the service registry. Finally, the client uses the service description to bind with the service provider. [3], [12]

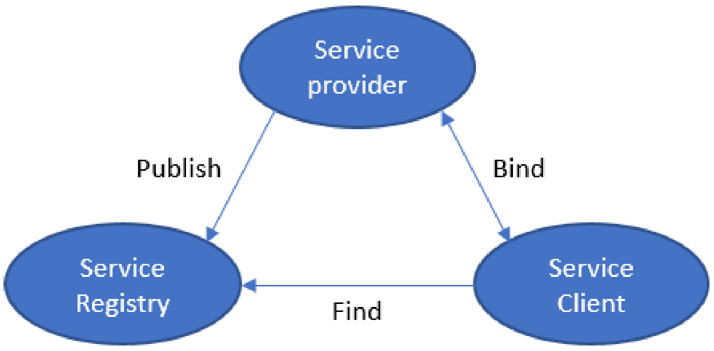


Figure 6. Basic SOA architecture.

To use services effectively, some of the characteristics need to be taken into account. By the means of effectiveness, the SOA and services need to be coarse-grained, interface-based designed, discoverable, single instance, loosely coupled, reusable, autonomy and asynchronous. The aim of coarse-grained services is to enlarge the functionality and operability with larger data sets. Interface-based design is one of the key factors in the SOA. It helps the implementation of a common interface by using multiple services. The idea behind discoverability is that services are found at both design time and run time. Single instance means that each service is a single instance with which multiple clients can communicate with. Loosely coupled services are implemented to be services that are connected to other services and clients using specific methods. Instead of communicating directly, a client and services use operations as a communication method. This procedure avoids any direct technical connections between a client and services. Typically, services use an asynchronous message passing approach. Additionally, the SOA is characterized by an abstraction. An abstraction is an important characteristic of the SOA by easing the understating of the complex systems and hiding the details of the implementation. By an abstraction, services of the system can be shown as black boxes by hiding the internal logic that can be accessed through an interface. Table 2 concludes the requirement for SOA. These requirements are later used to evaluate the proposed SOA. [1], [12], [13]

Table 2. Requirements for a SOA

ID	Requirement
REQ1	Loose coupling
REQ2	Autonomy of services
REQ3	Abstraction
REQ4	Reusability
REQ5	Discoverability
REQ6	Coarse-grained
REQ7	Interface-based

To summarize, the SOA is an enhancement of the CBSE. An individual service can be seen as a single component. They both aim at providing a base for loosely joined and vastly interoperable SW architecture enabling efficient SW development. There is no clear line between the SOA and the CBSE. Compared to the CBSE, there are two main points: services need to be publicly accessible and services need to be mostly independent from implementation specific attributes.

2.3. Software Testing Strategies

A test strategy is a systematic method which is used to select and generate different tests to be included in a test suite. Test strategies can be divided into three different categories: Behavioral test strategy, structural test strategy and hybrid test strategy. [14]

Behavioral testing or black-box testing is a testing method which is done under a behavioral test strategy. Black-box testing is based on requirements and is done in total ignorance of the construction of the object. There is no knowledge of the structure of the system or the component inside the box. Thus, the tester only wants to know how the software behaves, not how the software does it. The functional aspects of software systems are checked by black-box testing and the primary aim is to uncover errors and validate software. There are several types of black-box test types from which black-box testing can be divided into two best known methodologies: equivalence class partioning (ECP) or boundary value analysis (BVA). ECP is a testing technique where input values are divided into valid and invalid input partition, and from each partition a representative is selected to be a test data. On the other hand, BVA is a technique which is based on testing on the boundaries of various partitions. [14], [15]

Structural testing, also known as white-box testing, is a testing method which is done under a structural test strategy. The logical aspects of a software system are verified by white-box testing [15]. White-box testing demands full access to the structure of the system under test (SUT) and it is applied at the early stages of the testing process. The aim of white-box testing is to exercise a specific set of conditions, loops or paths. The knowledge of the structure of the SUT is the biggest difference between black-box and white-box testing. [14]

Hybrid testing or gray-box testing is a combination of black-box and white-box testing. Gray-box testing combines the benefits of both black-box and white-box testing, but it cannot execute the whole white-box testing because the inaccessible nature of the source code. Gray-box testing is said to be the best approach for functional or domain testing. Usually, unit and low-level components are tested using structural testing whereas behavioral testing is used to test big components and systems. Hybrid testing is suitable at all levels. [14]

As a summary, the test strategy is chosen according to the nature of the object that is tested, the nature of bugs in the object and the state of the knowledge of the structure. [14]

2.4. Model-Based Testing

Model-based testing (MBT) is an automation of a black-box test design. The main difference compared to the usual black-box testing is the creation of models [16]. MBT is a testing method that aims to automatically generate test cases from a design model which describes the functionality of a SUT. As a result, it is possible to automatically generate a large number of test cases from the SUT and there is no need to do the test cases manually. European Telecommunications Standards Institute (ETSI) has specified MBT as the umbrella of approaches that generate tests from models [4]. [17]

The MBT process consists of three main parts, modeling, test generation, and test execution. Figure 7 shows the MBT process. The modeling phase models the behavior of the SUT that is based on the predefined system requirements. The model is assumed to have knowledge of the input and output data of the SUT. The input data is used for executing the SUT and the output data is used for the validation purposes. To be efficient, models need to be described at a relatively high abstraction level. After the modeling phase, the test generation takes place. Test generation is based on model traversal where test design algorithms are utilized for generating test scripts from a model. In the end, test execution takes place. Test execution can be accomplished either online or offline. Online testing is generated step by step using the SUT output information whereas offline testing generates tests first and then executes tests separately. [18], [19]

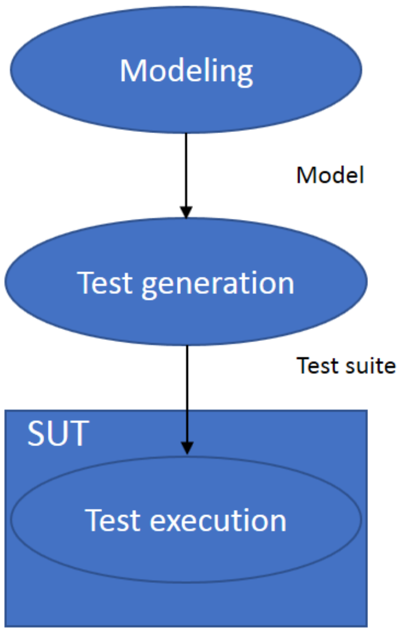


Figure 7. MBT process.

There are various MBT tools which can be identified in the three main types of licensing: commercial, open-source and self-made. Usually, commercial MBT tools offer the best support and availability by providing the simplest and the most customer friendly interfaces for modeling and editing. To avoid licensing fees, there already are some open source MBT tools that have an ability to modify the tool for personal needs. This kind of MBT tools might be the best choice when beginning to adopt the MBT testing process. The third type, self-made MBT tool, is a tool that is designed for a specific usage and need. [20]

According to [16], there are four main approaches know as model-based testing:

1. Generation of test input data from a domain model.
2. Generation of test cases from an environment model.
3. Generation of test cases with oracles from behavior model.
4. Generation of test scripts from abstract tests.

In the first approach, the model includes the information of the domains of the input values and the test generation implicates the specific combination of subsets of those input values generating test input information. In the second approach, the expected usage of the SUT is described by using several models. The difference between the second and first approach is that the second approach does not model the behavior of the SUT because the generated use cases do not define the excepted outputs of the SUT. The third approach uses oracle information to see if the output values are correct. This approach is somewhat complex because the test generator needs to know adequately the behavior of the SUT to be able to tell the output values. The final approach supposes that a general description of the test case is given and it concentrates on converting that test case into a low-level executable test script. [16]

2.5. Software Development Methods

To be able to develop a system effectively, a well-formed software development lifecycle is used. The commonly used software development lifecycle models are waterfall model, iterative and incremental model, prototyping model, spiral model and agile methods. [21]

2.5.1. Waterfall Model

The waterfall approach is a traditional approach that is used both in small and big projects. The basic idea is to enable structured software development by executing sequentially a series of development activities. The waterfall method contains such development activities as requirements, design, implementation, test and support. In the waterfall model, the next phase will start when the previous phase is fully finished. For example, the design part will wait until the requirements are decided. Because of this, the waterfall approach is quite slow. It is almost impossible to accomplish bigger projects without making any changes in the previous parts. Thus, making changes in the previous parts means that the entire process needs to be started at the beginning. Even today, this method is used with small projects where the process flow is systematic. There are no problems when the project works as planned, but when the project is more complex, some other approach needs to be used. [22], [23]

2.5.2. Iterative and Incremental Development

Iterative and incremental development was created due to the problems found in the waterfall method. The aim of the iterative and incremental model is to develop a software system incrementally by taking an advance of the previous steps and knowledge that is being learned during the development process. The learning comes from both the development and from the use of the system. The key is to start with the simple implementation of subsets of the software requirements and iteratively develop more and more advanced versions until the system is fully implemented. This method can be divided into five steps: requirements, specification, architectural design, implementation, and maintenance and retirement. In contrast to the waterfall method, the previous steps can be modified without interrupting the entire process. [23], [24]

2.5.3. Spiral Model

Spiral model is a software development method that has used waterfall method as an example. The spiral model has been developed based on the refinement of the waterfall method. It is close to the iterative and incremental model concentrating more on the risk analysis. The key feature of the spiral model is that it creates a risk-driven approach to the software process. The spiral model is usually used when the risk evaluation and costs plays important role, requirements are complex or significant changes are expected. The spiral model has four main phases: planning, risk analysis, developing (engineering) and evaluation. The entire process repeats these phases in iterations. The process starts with the planning phase. All the requirements, developments, integration and tests need to be planned. After planning phase, a risk analysis takes place. In this phase, all the possible risks are identified and the solutions are proposed. Next, the software is developed in the engineering phase. At the end of the engineering phase, testing is performed. The final phase, evaluation, evaluates the results of the project and the project continues to the next spiral. The spiral model allows the development of a software while decreasing the software development risks. [23], [25]

2.5.4. Prototyping

According to [26], there are three main approaches to prototyping: exploratory prototyping, experimental prototyping and evolutionary prototyping. The first approach, exploratory prototyping, is used when the problem is unclear. In this approach, initial ideas are used as a basis of requirements. The exploratory approach uses prototypes as a tool to find requirements in the early phase. Experimental prototyping uses prototypes to explore specific feasibilities or possibilities within the process development. The evolutionary approach is a continuous process that updates the needed requirements if needed. The main idea behind the prototype model is that a prototype is built despite the fact that all the requirements are not known. The aim is to provide a system with overall functionality. The main phases of a prototype model are requirement gathering, quick design, prototype building, customer evaluation, refining prototype and the final product. [23], [26]

2.5.5. Agile

Traditional approaches, such as the waterfall model, iterative incremental development and spiral model described earlier, have led to the concept of agile software development. Agile software development aims at enabling the development of runnable software that is possible to validate by both stakeholders and end users. Agile software development indicates to a set of methods and processes that are based on the agile manifesto. The agility concept concentrates not only on the domain architectures, but also on the modeling and implementation of an application. Iterative and incremental development is usually part of an agile strategy. [27]

The agile manifesto describes needed actions to develop software. There are four main statements that put confrontations to inspection. These confrontations are [27]:

- Individuals and interaction over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

According to the agile manifesto, the items on the left are more valuable than the items on the right. In other words, the main principles of the agile manifesto are to embrace changes and refactoring through the development process. The process is measured by the means of working software, where the developing process is iterative and is delivered in small increments. The first statement of the agile manifesto states that a team should define its own development process that is suited best to its specific actions. The main point of the second statement is to keep focus on delivering runnable software. The diagrams and software need to be up-to-date all the time. The third statement instructs to allow the customers to participate as much as possible in the application development, and the fourth statement encourages to be flexible. [27]

The best known agile techniques are extreme programming (XP), test-driven development (TDD), feature-driven development (FDD) and scrum. [21]

- XP is a technique where two developers share a terminal and execute the application together. The approach is based on simplicity and aims to minimize errors.
- The idea behind test-driven development is to first implement the tests and based on them, the application is developed.
- Feature-driven development consists of two main stages. The first stage concentrates on discovering the list of features and the second stage consists of implementing the discovered lists of features. The first stage

defines the quality of work; therefore, the first stage can be said to be the crucial stage. [28]

- Scrum is the most used agile software technique in the software industry. It is an iterative, incremental and empirical process that is used to manage and control the development of a project. The aim of scrum is the ability to react to the requirement changes. Scrum consists of three main roles: product owner, scrum master and scrum team. The main task of the product owner is to create priority based on a list of requirements, backlog. The scrum master leads the whole process and the scrum team is responsible of maintaining the process during each sprint. A duration of a sprint is from two to four weeks. [29]

The popularity of the agile methods has been growing over the last years due to lower costs and increased quality they provide. According to [27], there is a clear link between the model-driven software development (MDSD) and the agile techniques. MDSD can give support to agile techniques through domain knowledge and provide help through the separation of domain architecture and application development. To summarize, agile methods offer the biggest benefit when the environment is volatile. Agile methods aim to collaborate closely with the customer to be able to offer effective delivery and realize the risks.

3. MODEL-DRIVEN SOFTWARE DEVELOPMENT

MDSD has been taking a bigger and bigger role in the programming world and it is a constantly developing area. The idea of MDSD is to focus on models in software development instead of computer programs. MDSD offers an effective approach compared to a ‘basic’ programming language by offering completed, reusable components and frameworks. MDSD aims the focus of the software development more to the problem domain over the implementation by raising the abstraction level. The main goals of the MDSD are [27]:

- Improvement in development speed.
- Enhancement in software quality using automated transformations and modeling languages.
- Growing reusability once modeling languages, architectures and transformations have been specified.
- Enabling programmability on a more abstract level using modeling languages.
- Innovative environment in the engineering, technology, and management fields.

There are several ways how MDSD can be realized. In this thesis, Model Driven Architecture® (MDA) and domain-specific modeling (DSM) are presented to support MDSD [30]. Moreover, MDA and DSM support each other.

Chapter 3 is structured as follows. First, MDSD with MDA approach is introduced including metalevels, abstraction levels and the differences between diagrams and models. Second, MDSD with DSM approach is studied including DSML development processes and DSML designing guidelines. Third, the modeling tool MetaEdit+ and its concepts, which are used to create a DSM solution, are introduced.

3.1. MDSD with MDA

The Object Management Group’s (OMG’s) MDA is an approach to support MDSD. MDA emphasizes that the system must first be modeled before it can be fitted to the final execution platform. MDA is used to describe the usage of the models within the software engineering process. By using models, MDA drives people to understand complex ideas. MDA aims to use the system models efficiently in the software development process by supporting the reuse of models. There are four principles that underline OMG’s view of MDA [31]:

- Models are expressed in a well-defined notation to give an understanding of the systems. MDA drives to shift the focus of the SW development from the technology domain to the problem domain.
- A set of models are used to build the systems. These models are organized into an architectural framework of layers.

- A formal support for the models in a set of metamodels is the basis for the automation through tools. Using the tools, models can be transformed to a code. The aim of a model-to-code transformation is to increase speed and reduce human errors.
- To accept the model-based approach, it requires industry standards to provide openness to consumers and to enhance the competition among suppliers.

The OMG has defined a set of metalevels and levels of abstractions to support these principles. The following sub-chapters contain the definitions of metalevels and levels of abstractions as well as comparison of diagrams and models. [31]

3.1.1. Metalevels

This chapter contains the definitions of models, metamodels, meta-metamodels and mega-models. In addition, metamodels are discussed more detailed, because it is the most important level in this thesis. Metamodeling is a needed and one of the most important single steps in MDSD because it is a process of analyzing a domain. In theory, there could be an infinite amount of metalevels, as each metalevel can be described by using a higher metalevel [32]. OMG defines the four metalevels to describe metamodeling to prevent this endless loop. Figure 8 shows the relations between models, metamodels and meta-metamodels [27]. According to figure, metamodeling can be seen happen in the three levels M3-M1. From these levels, the models describing the domain itself are created. This four-layer architecture is a popular example supported by Meta Object Facility (MOF). [27], [33]

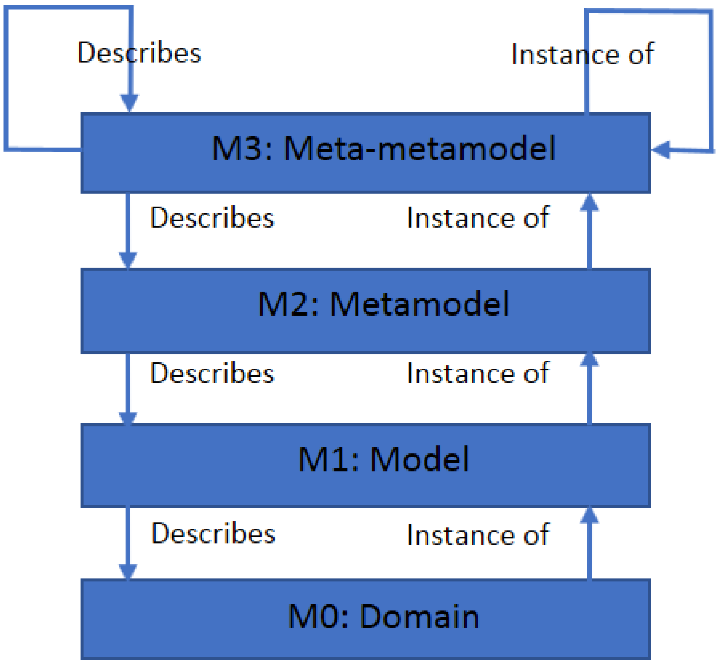


Figure 8. Four-layer metalevel architecture.

M3 layer is the meta-metamodel layer and it contains all the metameta-data. Meta-metamodel is an instantiation of itself and basically it defines itself. In this thesis, GOPPRR (Graph Object Property Port Relationship Role) is used as a metamodeling language to define metamodels, and it will be discussed later in chapter 3.2.3. M2 layer is the metamodel layer and it contains the metadata. Metamodel is an instance of meta-metamodel and it defines the language for specifying a model. In abstract way, metamodel defines the structure of models and modeling language (abstract syntax and semantics), as well as their relationships, constraints and modeling rules. M1 layer contains the model that is an instance of a metamodel and it describes a domain that will be implemented. Finally, M0 layer contains the domain that is an instance of a model. As stated before, the semantics, concrete syntaxes and rules are defined in metamodel layer. Therefore, the metamodel layer is the most crucial layer in this work. Thus, metamodeling is discussed more in the next paragraph. [27]

As mentioned, a metamodel defines the modeling language. According to [33], modeling language can be defined as "… a set of all possible models that are conformant with the modeling language's abstract syntax, represented by one or more concrete syntaxes that satisfy a given semantics." An abstract syntax describes the vocabulary concepts provided by the language and how they are used to create models. Metamodel defines a one abstract syntax. However, there could be multiple concrete syntaxes. The concrete syntax refers to its notation and it can be divided into two main types: textual and graphical notations. For example, the model can be expressed by using codes (textual) or diagrams (graphical). The abstract syntax needs to be defined to have a good balance between simplicity and expressiveness. In this thesis, a graphical notation is used as a concrete syntax. The semantics is needed because the language is often context dependent. The semantics can be depicted as constraints, and it blocks its user from creating models that break the rules and the orchestration of its elements. The semantics needs a proper tool support. To have a sufficiently specified modeling language using a metamodel, these key parts (syntaxes and semantics) of the language need to be modeled. If graphical modeling languages are used, abstract syntax is specified first. After defining an abstract syntax, a concrete syntax is defined as a mapping of graphical notation onto the abstract syntax. [34]

When a large number of models need to be handled at the same time, a concept of mega-model rises. A mega-model is a conceptual framework whose focus is to model a large-scale software evolution process. Basically, mega-modeling aims to solve the problem that lies in the large-scale software business: how to handle large entities that include multiple different models. Behind the four-layer model, there is a clear concept of OMG's MOF. In mega-modeling, there is not such a clear base. There is only a concept or an idea of how those model volumes can be dealt with by using the expedients that are commonly used in the modeling world to present the relations between different models. The aim of mega-modeling is to deal with the models, metamodels and their relations, and also to provide the possibility of defining relationships between models. In the concept of mega-modeling, the models can be divided into metamodels or meta-metamodels. [35]

3.1.2. Levels of Abstraction

To achieve the independence from SW application platform as well as to achieve longevity in the software development, MDA defines three levels of abstractions [36]:

- The computational independent model (CIM): The focus is on the environment of the system and the structural details concerning the implementation platform are hidden. Platform in this case is the set of different technologies and subsystems that provide the needed functionality.
- The platform independent model (PIM): The system is described from an independent point of view of the platform. The abstractions of one or more platforms are captured by hiding the specific data of the platform.
- The platform specific model (PSM): Represents the system and its platform specific data. The details of the specific platform and specification of PIM are combined.

Often, MDA is seen as a process where executable software systems are generated from formal models starting with CIMs extending them to PIMs to be adapted into PSMs and further resulting in source code. Partition between PIMs and PSMs is one of the main concept of the OMG's MDA. This method works as a bridge covering the traditional gap between human-readable requirements and source code [28]. [27], [36],[37]

3.1.3. Diagrams vs. Models

Many people consider diagrams and models as synonyms although there is a clear difference between them. In this context, diagrams can be considered as a part or an aspect of a model. Usually, diagrams are visual representations, e.g. shapes, lines or nodes, describing the system, and a model is the whole description of the system that is machine-readable. Thus, when the modeling process is going on, it is not only modifying diagrams. Models can also be expressed in other forms, such as matrices, tables, trees or maps. [30] In this thesis, the models are part of the M1 layer described in chapter 3.1.1.

To achieve a model-centric environment instead of just drawing diagrams without any constraints or rules, at least the following points need to be embraced [38]:

- Consistency: When the model is the base for all the modeling perspective, there should not be any conflicting perspectives. This is due to the fact that all the perspectives are extracted and governed from the same model source. Therefore, if the perspective will change, the model will also change, and vice versa.
- Collaboration: Using models that are well-defined allows others to modify and add elements to models preserving the functionality of the model.

Even though the model can become large and complex, the model is still consistent.

- Visibility: Using models, the complex mechanisms can be hidden and expose only the needed information through diagrams to ease the understanding of the idea behind the model.
- Automatic perspective generation: Diagrams, matrices, reports and many other perspectives can be directly extracted from the models. Particularly, automatic code generation is a beneficial approach in SW development area.

3.2. MDSD with DSM (Language)

Models and modeling languages are used when the abstractions of a software system are created. Some software systems demand very specific design modeling. Generally, software modeling can be divided into two categories by means of the usage purpose of modeling languages: general-purpose modeling languages (GPML) and DSML. GPMLs are suitable for many software design problems in many different domains. Using a general set of software concepts, these languages focus on describing multiple software systems at the same time. One of the well-known GPML is Unified Modeling Language® (UML) which is standardized by the OMG. One way to use UML is to describe software system using an independently separated object-oriented concepts from the programming language. DSM and DSML have a big role in this thesis so they are described more closely in the following chapters. In general, DSM and further DSML are used to keep the focus on one specific, restricted application domain. Using these concepts, suitable modeling elements for the specific domain can be implemented instead of defining general standards. [39]

In DSM, the purpose is to enable a modeling language that is suitable for a certain need and that the modeling, with the created modeling language, is simple and efficient. DSM with the help of its tools provide precise design analysis and automatic code generation to achieve better system quality [40]. DSM is a product of an evolution of MDSD. DSML has been developed due to the next level of abstraction beyond current programming languages. The modeler first defines the metamodel including the modeling language and then the possible rules and constraints are defined to guide the modeling itself. The important thing is, that the modeler can define exact the kind of metamodel and modeling language that is needed. The key is to create a (meta)model from which the final code can be generated by using high level specifications that have specified the solution directly using the problem domain. This is reached by using DSML that allows the developer to focus on the solution rather than the technical implementation of the solution by following domain abstraction semantics. [41], [42], [43]

DSML formalizes the application structure, behavior and requirements using a specific domain. DSML is often considered as a graphical language which interprets the ideas and logic using visual diagrams. There is a wide range of possible domains, e.g. technical domain, user interface, functional, business etc. The smaller the domain the easier it is to automatize. Basically, the modeling language, code generator and framework code are domain-specific and are fully under the control of their users. Often DSM and DSML are used as a synonym. To summarize, there are two main things at which DSM aims. First, to raise the level of abstraction using a language that is created to solve a problem using concepts and rules. Second, to develop the final product by using a chosen programming language or other form from the used specifications. [41], [42], [43]

By using a certain language, a model of a solution, i.e. a specification model, is created, and which contains all the concepts and rules from the problem domain. The division between models, code generator and framework code is important. Generally, the models are used only to describe the behavior of the product while the framework ensures the interface for the target platform and programming language. The framework also provides a specific set of services to which the code generator can interface. Finally, the way how information is extracted from the models and transformed into code is specified with the code generator. The code and framework is linked together and as a result, it is executable without any additional manual work. [41], [42], [44]

Why choose DSM over the other possible choices? Maybe the most significant benefit of DSM is the increase in development productivity. For example, Nokia [45] shows the increase of productivity gains of 5 to 10 times of traditional manual practices. According to test results, DSM makes the specifications easier to read, understand, remember and validate. [46]

3.2.1. DSML Development Process

Developing a DSML the developer needs to have a valid knowledge about the domain and the concepts behind the modeling language. Usually, DSML development process is a collaboration between domain experts and engineers that develop the modeling language for that domain. Figure 9 shows the development process [47]. The process starts by capturing the requirements of DSML and requirements of the system where the DSML is being developed. After the requirements are clear, based on them, the concrete syntax or abstract syntax is identified depending whether the language is graphical or not. If the modeling language is graphical, the abstract syntax is defined first and the concrete syntax is defined second, and vice versa if the modeling language is textual. After defining the syntaxes, the language semantics are attached to them. Finally, DSML is verified by the domain experts based on whether the set requirements for the DSML are fulfilled or not. This kind of iterative and incremental process for defining DSML grammar and its semantics is quite a challenging task [48]. According to [47], the aim is to simplify and automate DSML development using three targets:

1. Capturing the concrete syntax as end-users perform modeling tasks in their domain.
2. Deducing the abstract syntax from the concrete syntax and model instances.
3. Attaching the semantics to the abstract syntax.

[49] and [50] added four phases to reach the targets above:

1. Identifying abstractions and how they work together.
2. Specifying the language concepts and their rules (metamodel)
3. Creating the visual representation of the language (notation)
4. Defining the generators for model checking.

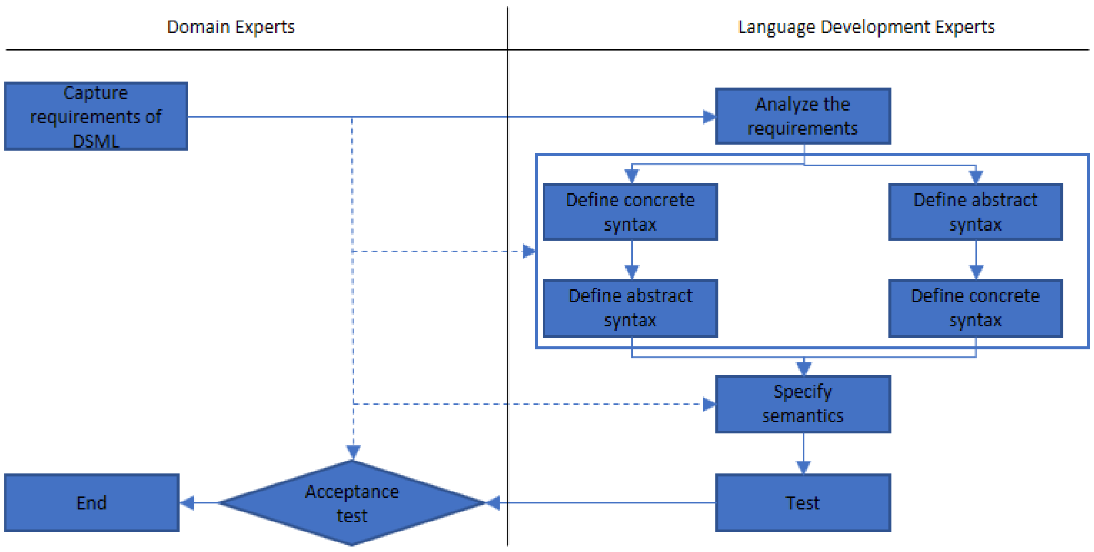


Figure 9. DSML development process.

The following steps are typical when developing a DSML [51]:

- Analysis
- Implementation
- Use

The first step, analysis, finds the problem domain and then collects all the essential knowledge in this domain. Then, the knowledge is clustered in a smaller semantic notions and operations. The final part of the analyzing step is to design a DSML that briefly describes applications in the domain. The second step, implementation, forms a library that implements the semantic notions. The language concepts and their rules are specified. After that, an assembler is implemented to translate DSML programs to a sequence of library calls. The last step is to create DSML programs to all the needed applications and to assemble them. To summarize, the analysis step builds up the core of the application domain. The implementation step is a so called working step where the theory behind the analysis step is converted into a real DSML. Finally, DSML is put into use. [51], [52], [53]

3.2.2. DSML Designing Guidelines

The previous chapter presented the development process of DSML. This chapter focuses on the DSML design guidelines from a software designer point of view:

What should be considered when designing DSMLs? DSML design is an iterative process, therefore the guidelines presented below are part of every step discussed in the previous chapter. According to [54], DSML design can be divided into five categories: language purpose, language realization, language content, concrete syntax and abstract syntax. It needs to be noticed that the line between these categories is blurry, and the guidelines could overlap. As a language developer, the balance between the guidelines needs to be found. Some of the guidelines are quite general and obvious, but at the same time, it must be remembered that the simpler the language the more effective it usually is. The following paragraphs will describe each category more closely. [54]

Language purpose category covers the design guidelines in the early phase of the language designing and development process. This category could be divided into three guidelines [54]:

- The usage and necessity of the language need to be defined and the aim should be clear.
- Asking questions: Using simple questions like “Who is going to model in the DSML, when and for which purposes is the model used?” will help to notice the necessary decisions.
- Usually DSMLs are designed for certain purposes, so the language should be consistent.

There are multiple options to realize the new domain-specific language. For example, it can be implemented from a scratch, existing languages can be utilized and a graphical or a textual representation can be used. This category could be divided into the following guidelines [54]:

- It is important to investigate carefully whether the graphical or textual realization fits to the usage. Pros and cons need to be listed, after which the decision can be made.
- If it is possible, the utilization of an existing language is recommended. Reusing an existing language saves a great deal of working hours.
- Even though the existing language is not reusable, the language definitions are still often valid for reuse.

The language contents will vary a lot among the languages so this is divided into very basic guidelines [54]:

- The importance of simplicity cannot be overemphasized. The simpler the language, the smaller probability there is to encounter errors. The content that is not necessary can be left outside.
- The fewer language elements there are, the easier it is to understand the language. Thus, the number of language elements needs to be limited.
- The ineffective language elements and conceptual redundancy need to be avoided to make the language efficient.

Concrete and abstract syntaxes were briefly discussed in chapter 3.1.1. Both syntaxes will be discussed more; first the concrete syntax and then the abstract syntax. The importance of the concrete syntax is huge when designing a DSML. The following guidelines encapsulate the concrete syntax [54]:

- Adoption of existing notations: It is recommendable to use existing formal notations rather than invent new ones if the domain experts already have notations.
- Usage of descriptive notations: A descriptive notation contributes both learnability and comprehensibility of a language.
- Distinctiveness of elements: The elements need to be understandable. For example, in graphical DSMLs, different elements need to have different representations that depict enough syntactic differences (colors, shapes etc.).
- Compactness and comprehensibility: Using comments, clear hierarchy, the comprehensibility of notations, and using the same style everywhere makes the language easier to read.

The guidelines for abstract syntax could be presented as follows [54]:

- The structure of the abstract syntax should follow closely the concrete syntax. Thus, the elements that differ in the concrete syntax need to have their own abstract notations.
- For simplicity, the layout of the programs should not affect their semantics.
- Using the language, the system should be able to be decomposed into smaller pieces.
- DSMLs should offer an interface concept similar to the interfaces of known programming languages.

3.2.3. DSM Tool: MetaEdit+

Martin Fowler introduced the term “Language workbench” in 2006 [52]. Language workbench is a new category of tools. In this work, MetaEdit+ tool can be considered as a language workbench. MetaEdit+ is used to implement the domain-specific solution. Language workbenches were defined as tools which have their own environment that is created to help people define new DSMLs using high-quality tools to use DSMLs efficiently. Language workbenches offer an opportunity to custom editing environment to that language. Language workbenches, like MetaEdit+, also offer a support for diagrammatic languages and graphical representations. These tools allow users to define a DSML in three key parts: schema, editors and generator. Also, language workbenches support syntax highlighting, code completion and a debugger [55]. Maybe the biggest advantage of language workbench is that it is possible for non-programmers to program. Even though language workbenches are quite a new concept, they have a great potential to become a major tool in the software development field. To summarize, a language workbench is a specialized integrated development environment (IDE) not only for specifying and constructing DSMLs but also to enable environment to write DSML scripts that combine the editing environment and the language for writing. [52]

DSM tool frameworks are developed to minimize the effort of developing tools support for a DSML. Usually DSM frameworks consist of DSM based tools that are necessary to develop customized tools supporting the development of applications. DSM framework provides frameworks for designing, editing, validation, analyzing and testing. One of the basic idea of DSM tool frameworks is to reuse the same generic tools for many domains in the modeling point of view. The support for a modeling framework is central. The basic tool support for modeling and support for automation is defined by DSM tool frameworks. In this thesis, MetaEdit+ is used as a DSM tool framework and it will be discussed next. [39]

MetaEdit+ is a platform-independent graphical language workbench for DSM that can be considered as a next generation computer-aided software engineering (CASE) tool. It is a tool set for creating and using modeling languages and code generators. It provides a flexible environment that is focused on specific domains and allows building models and generators without having to write a single line of code. By configuring the generic tool set with metamodeling, MetaEdit+ offers tool support for metamodeling languages. GOPPRR metamodeling language is used to define the models. MetaEdit+ offers simultaneous use of multiple metamodeling languages. An object-oriented repository system is used to store the data of models. This repository allows multiuser activity and enables parallel data share. MetaEdit+ Workbench (MWB) version is used in this thesis. MWB integrates the language and generator development tools and ordinary modeling tools. Figure 10 shows the architecture of MetaEdit+. [56], [57]

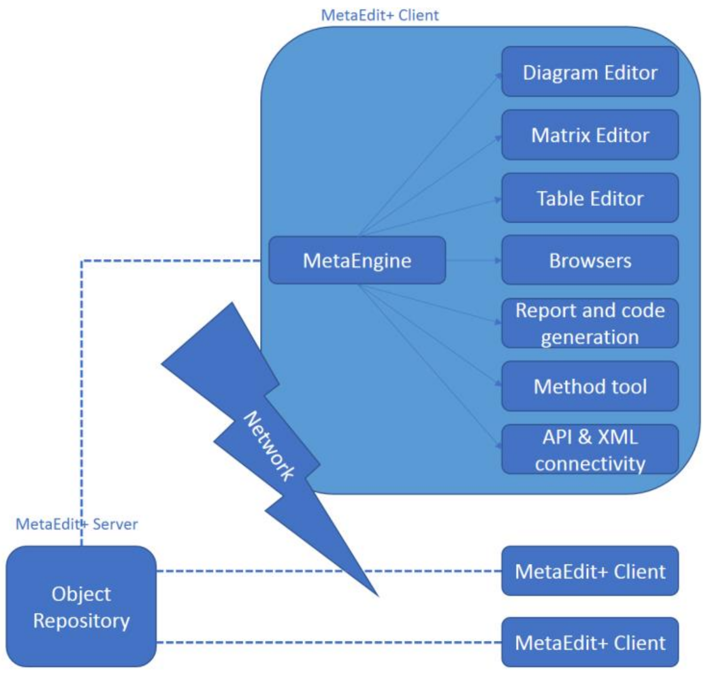


Figure 10. The tool architecture of MetaEdit+.

The MetaEdit+ environment supports multiple users simultaneously to use multiple tools. These tools provide a different view of the same objects. The environment also offers a consistency checking as well as providing several representation formats for the same design objects. MetaEdit+ offers an environment that aims at improving usability, flexibility and open nature of CASE. These aims have been the base for the tool architecture. The main principles of the tool architecture are conceptual modeling, layered data base architectures and object orientation. These principles make it different compared to the traditional CASE approach. MetaEdit+ supports generic CASE behavior for objects and relationships, including different modeling editors, browsers and property dialogs. Furthermore, compared to CASE approach, MetaEdit+ offers XML import and export capabilities and an API for data and control access to MetaEdit+ functions. [56], [57]

Modeling of a SW is executed by using diagrams, matrices and table editors. These three editors offer a different perspective of the underlying domain. The diagram editor is the main editor in MetaEdit+ because its graphical representation is a natural choice to work with visual modeling languages. Matrix and table editors are options for editing modeling data. [56], [57]

Report and code generation is one of the main blocks in the MetaEdit+ architecture (Figure 10). The main idea of code generation is that the generator goes through the design models, extracts data from them and presents it in some predefined format as an output. MetaEdit+ supports several generators that the user can choose from, e.g. C. MetaEdit+ Reporting Language (MERL) scripting language is used to define generators. MERL is specified for creating code generation definitions. [56], [57]

GOPPRR

In this thesis, metamodeling language GOPPRR (Graph Object Property Port Relationship Role) is used. GOPPRR is an own metamodeling language of MetaEdit+. The aim of GOPPRR technique is to give the method engineers the maximum degree of freedom. By using this approach, everything that is possible to define with the metamodeling tools is a valid technique. All the GOPPRR modeling concepts come directly from MetaEdit+. For example, Object in GOPPRR does not mean the same as the object defined in the object-oriented software. Next, the meaning of every capital letter of GOPPRR concept is briefly discussed. [58], [59]

The first letter of GOPPRR, G, stands for Graph. Graph describes only one graph type, e.g. use case diagram and state diagram. Graphs contain a certain number of objects and their relationships and they have their own properties. All the specifications and details of each graph type are modeled with a distinct metamodel. [59]

The next letter, O, stands for Object. The objects are the key elements of the design describing the key concepts of a modeling language. Objects are such elements that are used frequently and commonly reused, e.g. messages and states. [59]

The letter P in GOPPRR represents the Property. Property defines all the attributes that characterize the four other language concepts (graph, object, role and relationship). Properties can show a different data type, such as string or Boolean. The second P stands for Port. It is an optional part of an object, to which a role, the second letter R in GOPPRR, can be connected. Ports are used when it is wanted to connect a role to a specific part of an object. [59]

The letter R in GOPPRR represents the Relationship. Objects associate with each other by using relationships. This concept handles the connectivity, such as association and inheritance, between the objects. The second R is the Role. Both Rs are linked together in such a way that Role specifies the lines and end-points of relationships. Roles define the participation of objects in specific relationships. [59], [60]

Among the previous concepts, there are several other language concepts that are important when working and modeling with MetaEdit+. Those language concepts are binding, object set, inheritance, aggregation, decomposition and explosion. For example, the aggregation and decomposition methods are used by collecting reusable elementary method types with the help of the concept Graph. By adding rules to properties, GOPPRR checks the model decency. These concepts are discussed later in chapter 4 when the modeling process has been started. [58], [61]

MERL

A code generation is one major part of the MDSD. Thus, MERL provides creation of code generators. MERL is used to define restrictions, rules and part of the symbols of the objects. Also, documentations and the reports of the models are done by MERL. MERL is an own object-based scripting language of MetaEdit+ that has its own syntax. However, this syntax is quite similar to, for example, C++. The created language allows to navigate through the models extracting them and generating the output text. Also, MERL provides a number of commands that enable various user interventions or execution of external programs. These commands are, for example, for-loops and if…else statements. All the code generators are done in the generator editor which provides functionalities such as traceability, debugging and code highlighting.

The semantics of DSML are defined in the code generator. The code generator of MetaEdit+ goes through the models and uses the pre-defined information of the models to generate reports and documentation. The code generator consists of a set of reports which all can call other reports. Each graph must have an own report. [30]

The biggest benefit to use MetaEdit+ generator is the possibility to integrate the metamodels with the code generator editor. Therefore, the models and generated output are always in sync. This results in developing the language and the generator definition in an agile way. [60], [62]

4. SERVICE-BASED MODELING AND DSML

The current component-based point of view in the modeling field is somewhat problematic due to the construct of the large-scale enterprise. Regardless of whether or not the construct was component-based or service-based, the construct in the SW modeling perspective focuses on the different SW teams. Every SW team has their own responsibility area of which they are in charge. Now, in the component-based structure, these teams are focused on the different SW components, but the contents and scopes of the components are becoming too extensive to fulfil the idea of an efficient way of working as a SW team. A service-based approach in modeling could offer an alternative to solve the rising problem of inefficiency among the component-based modeling. Instead of being responsible for the different SW components, in the service-based approach, every SW team is in charge of one or more services, which includes, for example, interfaces, development and testability. When a new feature comes from backlog, it needs to be planned in such a way that the impacts in the different services are recognized. After a successful planning operation, every SW team will develop and test their own services and finally the new or updated version of a service is put into a trunk. Based on the problem stated above, a following hypothesis is formed:

- H1: A service-based approach might solve the problem that the current component-based approach has faced.

One of the aims of this thesis is to validate this hypothesis. The expectation is that using service-based structure, the SW functionalities can be divided in more organized way to improve the quality of the SW and reduce time in modeling.

In this thesis, an object-oriented based specification and description language (SDL) has been taken advantage of. SDL is introduced in [63]. Some of the SDL characteristics, such as the hierarchical language structure and the graphical presentations of the symbols, have been used when the modeling language concepts of this work have been developed.

This chapter is structured as follows. First, the used domain and the current SW structure are introduced. Based on the domain, a service-based solution is implemented. Because the used domain is large and it contains multiple different functionalities, to save time, one features of that domain is modeled. Second, all the phases of developing the DSML are explained in detail. Third, the service-based models are created using defined DSML and the code generation is introduced.

4.1. Introduction to Example Feature

In 2003, the International Standards Organization (ISO) and the International Electrical and Electronic Commission (IEC) approved the RapidIO Interconnect Specification in the BTS area. Since then, RapidIO is the only authorized system of interconnection technology. RapidIO is an open standard for a high-bandwidth, packet-switched interconnection supporting data rates up to 60 Gbits/s. There are both parallel and serial versions for RapidIO, and in this work the serial RapidIO (sRIO) is investigated. One advantage in using the RapidIO is that it is suitable for embedded systems because it uses low-voltage differential signaling technique to minimize power usage. The RapidIO protocol consists of three layers: logical, transport and physical layer. The logical layer controls the end-to-end interaction between endpoints. It also defines the protocol, packet formats, initiating signals and ending signals. Transport layer provides a path that enables the transmission of information to each node. The physical layer defines the packet transmission, information control and electrical characteristics. The three-layer structure of RapidIO increases the capacity of the product. [64], [65]

RapidIO includes both serial and parallel versions. The feature in this thesis define multicasting operations in the serial RapidIO and the aim is to provide multicasting functionality to the system. Multicasting means the concept of duplicating a single message and sending it to the multiple defined destinations. In the sRIO systems, the capability to duplicating messages should scale with the number of end points in the system. The multicast is defined for switches only, because the number of the end points scale with the number of switches. The multicast mechanism has several goals that need to be fulfilled. For example, it must be simple, compact, robust, scalable and compatible with all the physical layers. The multicast operations have two control value types: multicast masks and multicast groups. In this work, multicast groups are studied. The multicast groups are defined as a set of target end points which all receive a specific multicast packet. Each multicast group is compound with a unique destination ID. A multicast mask is a value that decides the association between the multicast groups and the egress ports. Figure 11 shows the feature under design. [65], [66]

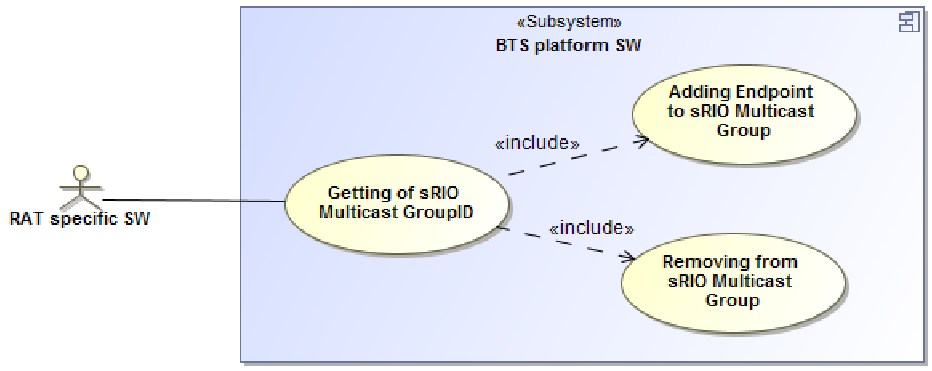

«Subsystem»
BTS platform SW
Adding Endpoint to sRIO Multicast Group
«include»
Getting of sRIO Multicast GroupID
RAT specific SW
«include»
Removing from sRIO Multicast Group

Figure 11. The feature under design.

4.2. Current Component-Based Structure

This chapter describes the current component-based structure of the SW inside the BTS system model. The current component-based structure utilizes both the services and the SW components, of which the SW components are the ones that are needed to be removed in the scope of this thesis. The current component-based BTS system model consists of RAT specific SW, known as client, and platform SW models. The platform SW models and RAT specific SW consist of architectural and functional models. The architectural model specifies the internal SCs as well as their provided and used interfaces to allow the communication between the components. The functional model focuses on specifying the interaction of components and which components are needed to complete the feature. The usage and the functional behavior of the system are defined in the functional model. Also, functional models place requirements to the interface models, which, for example, defines the type of operations. In the current component-based structure, the interface models are specified after defining the architectural and functional models. The interface models define each interface separately. Also, the interface model specifies the set of services that can be used via the interfaces that are provided by SCs. The architectural and functional models are created for each SC. In this work, features under design consist of all five models described above: platform SW architecture and functional models, interface models and system component architecture and functional models. Figure 12 shows the hierarchical structure of the BTS system model. [4]

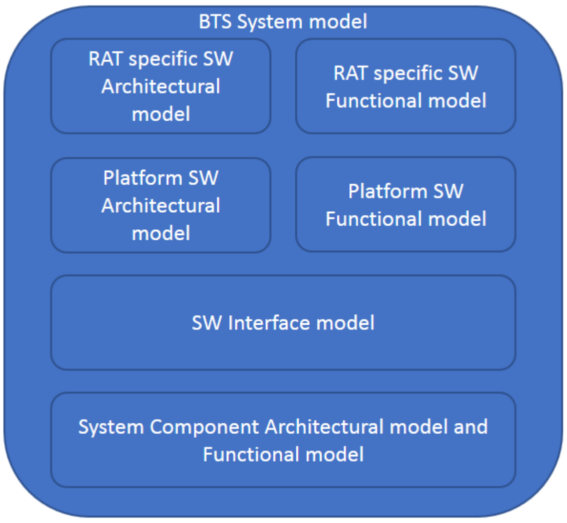

BTS System model
RAT specific SW Architectural model
RAT specific SW Functional model
Platform SW Architectural model
Platform SW Functional model
SW Interface model
System Component Architectural model and Functional model

Figure 12. The hierarchical structure of the BTS system model.

4.3. Proposed SOA

The agile SW development methods, that is introduced in chapter 2.5.5, are utilized to develop a service-based SW structure. The proposed SOA was developed in two to four weeks sprints. After the sprint, a meeting with the team, which includes SW specialists and line managers, was organized and the development proposals were discussed and listed. Based on those proposals, the proposed SOA was further developed.

The proposed SOA consists of three different abstraction layers. The service group level is the highest abstraction layer, the external service and service level forms the second abstraction layer and the interface description and micro-service description level compose the lowest abstraction layer. Each level is either a black box or white box depending the usage of the level. The black box view means that the user does not see the functionality inside the object or the level. On the contrary, white box view shows the internal functionality of the level. A specific graph type is defined for each level to depict the specific characteristics of each level. These graphs are listed below. The text *black box* or *white box* in the brackets denotes whether the level uses objects from the white box or from the black box view.

- Service group graph (black box)
- External service graph for a certain domain (white box)
- Service graph for a certain domain (white box)
- Interface description graph (black box)
- Micro-service description graph (white box)

Because of the properties of the modeling tool, black box view and white box view have been divided into two parts. The first part of the black box view describes only services for a specific service group as a black box and the second part of the black box contains a UML-like sequence diagram type of functionality to describe interfaces. The black box views contain the following objects: service, lifeline and requirement. The second part of the metamodel is called white box view and it contains all the objects that are used to model the behavior of the service and relations of the services. The white box view is also divided into two parts. The white box view includes the objects: micro-service, operation, interface, input, output, call, start, decision, task and end. As a result, in the proposed metamodel, there are four possible graph types: *BlackBoxI*, *BlackBoxII*, *WhiteBoxI* and *WhiteBoxII*.

Briefly, the service group graph is the uppermost abstraction layer that shows all the services of that service group. This graph is for external user purposes. The external user sees the overview of the service group graph, but not the details and functionalities inside the objects. From this graph, the access to external micro-service graph and micro-service graph is provided. In this thesis, the availability of the graphs and elements are in the client point of view. Therefore, external micro-service is external for the client and so on. External micro-service graph shows all the external micro-services and their relations through an interface. Furthermore, micro-service graph shows both the external and internal micro-services and their relations to other internal and/or external micro-services through an interface. This level shows more detailed functionalities of the domain. The micro-service graph is for modeler purposes because both internal and external micro-services and their relationships are depicted. External micro-service graph is meant more for the client purposes, because this graph does not show the internal functionalities of the micro-services. From these graphs, the access for interface description graph and micro-service description graph is provided. The interface description graph shows the message sequences between micro-services and the micro-service description graph models the actual behavior of the specific micro-service. Figure 13 shows the overview of the abstraction layers. Briefly, the main elements of the proposed SOA are service, micro-service and interface. A service is an implementation of a functionality that is delimited by a one interface. Micro-service is a service which functionality is limited into a smaller area. Interface is a defined entry point of a functionality provided by service or micro-service. Each of these graphs are discussed and explained in detail in the next chapters.

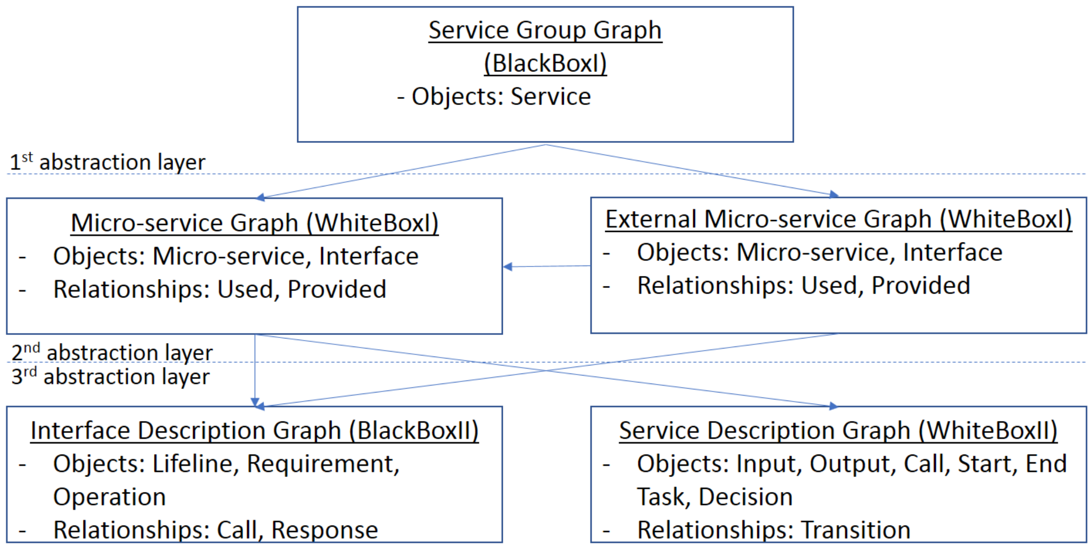

Service Group Graph
(BlackBoxI)
- Objects: Service
1st abstraction layer
Micro-service Graph (WhiteBoxI)
- Objects: Micro-service, Interface
- Relationships: Used, Provided
External Micro-service Graph (WhiteBoxI)
- Objects: Micro-service, Interface
- Relationships: Used, Provided
2nd abstraction layer
3rd abstraction layer
Interface Description Graph (BlackBoxII)
- Objects: Lifeline, Requirement, Operation
- Relationships: Call, Response
Service Description Graph (WhiteBoxII)
- Objects: Input, Output, Call, Start, End Task, Decision
- Relationships: Transition

Figure 13. The abstraction layers of SOA.

4.3.1. Service Group Graph

The aim of this graph is to give a client an overview of all the available services and their properties and to show the behavior of the service group as it is perceived by external users. The service group is divided into logical entities, services, of which each service includes specific functionalities. The service object is used to depict those services. Furthermore, the services are introduced in this graph, but there is no reference to its internal architecture. Thus, this level should not take a stand on its internal architecture. That is, this level is black box.

Figure 14 shows an example of a service graph that consists of three different services; *Service1*, *Service2* and *Service3*. The figure below also shows the pop-up window when the *Service3* is opened. The figure shows all the possible properties of the service. For example, the precondition and the description field are editable as a text, but the other fields will demand either existing or a new object depending on the property field. In this graph type, there are rules and restrictions. For example, service is the only object that can be used in this graph. Therefore, there is no possibility for using other objects than services. The full list of rules and restrictions of each graph are gathered in chapter 4.5.

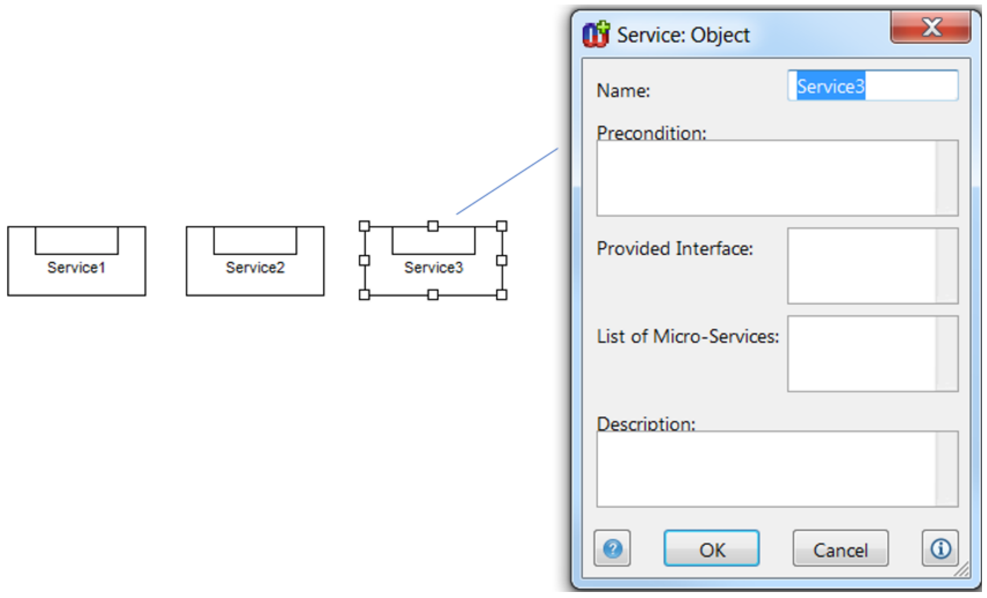


Figure 14. Overview of the service group graph.

4.3.2. External Micro-Service Graph and Micro-Service Graph

From the service group graph, it is possible to move either to external micro-service graph or micro-service graph. The aim of the external micro-service graph is to show the possible external micro-services that are available to the application. The internal micro-services are so called black box objects to the client. Hence, the client does not need to know what happens internally in the SW. As said, external micro-service graph is available for the client. Thus, the external micro-service graph shows the external micro-services and the relations between them through a well-defined interface. In addition, this graph type describes functionalities more detailed way without referencing to its internal architecture. On the contrast, micro-service graph shows all the micro-services (internal and external) and their relationships through interfaces. The micro-service graphs give references to its internal architecture. This layer also shows whether the interface is provided or used. The micro-service graph is meant for the modeler and the external micro-service graph for the client purposes, but the graph type does not exclude each other. In other words, the idea of this graph type is to show the micro-services and the relationships between them via well-defined interfaces. Both, the external micro-service and micro-service graphs, allow only to use micro-service and interface objects in the modeling purposes. All the other objects are restricted.

From the micro-service graph, there are multiple options where to continue. One can access the interface description graph by double clicking the relationship between the interface and micro-service. One can also access the interface description graph by choosing the right interface description graph on the pop-up window by clicking the interface itself. Therefore, there are two possibilities to access the interface description graph. The third possible access to the third abstraction layer from the second layer is from the micro-service. By choosing the observed micro-service, the micro-service description view can be accessed. From the external micro-service graph, there is no access to the micro-service description graphs. Therefore, to access the micro-service description graph, the micro-service graph needs to be accessed first.

Figure 15 shows an example of the micro-service graph and external micro-service graph. The figure shows that there are two external and two internal micro-services. Both external micro-services provide the interface *IF2* and use the interface *IF1* that are provided by the internal micro-services. The external micro-service graphs show only the external micro-services and the interfaces that are provided by them.

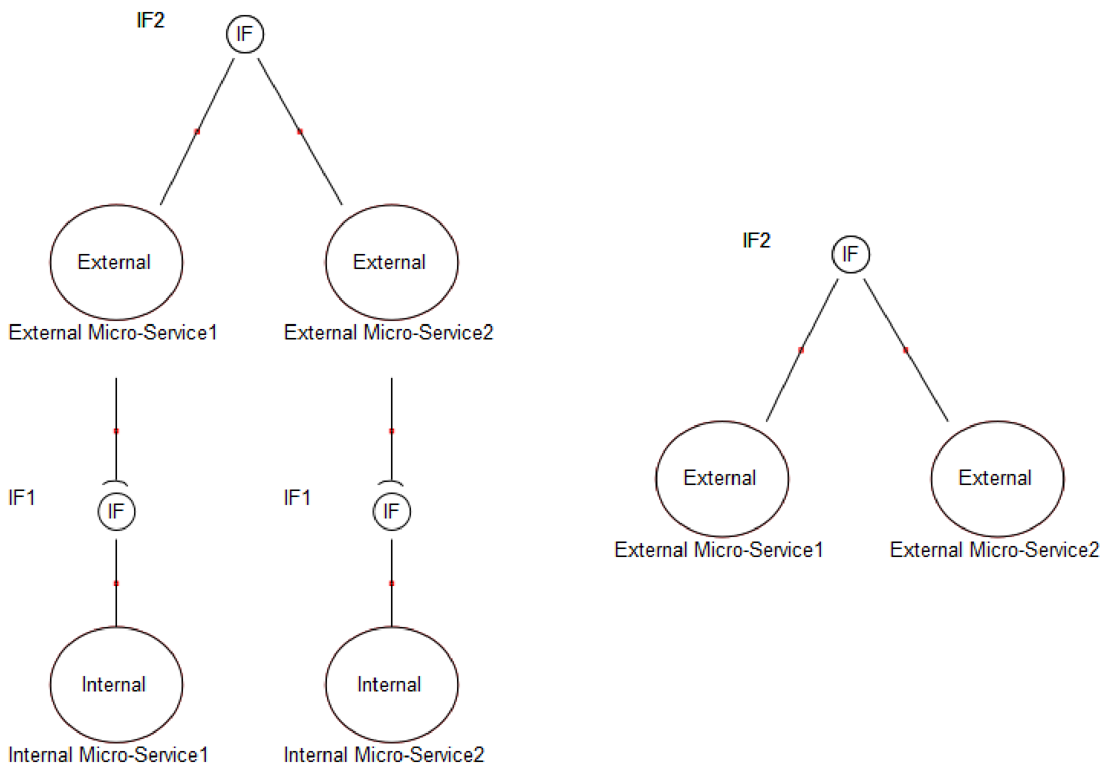


Figure 15. The micro-service graph on the left and the external micro-service graph on the right.

4.3.3. Interface Description Graph

The interface description graph shows the interface description to a specific micro-service. The aim is to show how the interface is performed to a specific micro-service. This graph defines the interaction between micro-services via interfaces. Therefore, interfaces need to be up to date. The interface description is usually based on the requirement; therefore, it is also shown in this graph type. The requirement shows the needed characteristics of the interface. In addition, the requirement where the interface is based on is for modeler purposes. On the other hand, this graph shows to the client the needed operations, which by the client can use the micro-service in question. This graph type uses lifelines that show the micro-service for which the interface description is done, and the interface itself. Usually interface description is request-response operation pair. In some cases, there is also some indication message(s). Call and response relations models the request-response operation pair.

Call and response can be synchronous and asynchronous; hence the operation type is also shown in this graph. Basically, this graph type defines the interaction between micro-services or client and micro-services via interfaces. Figure 16 presents an example of the interface description graph.

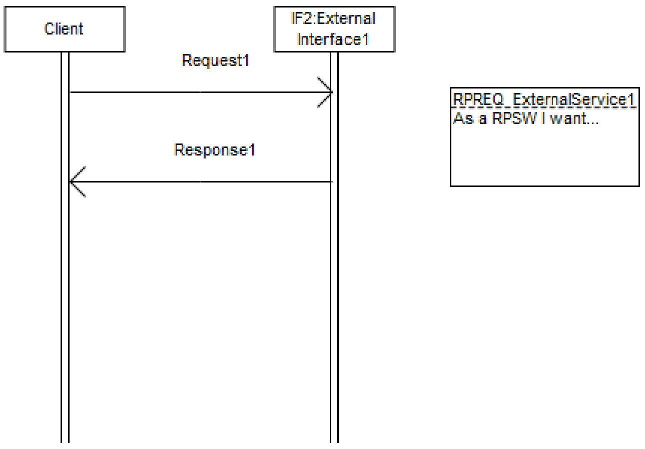


Figure 16. The interface description graph.

4.3.4. Micro-Service Description Graph

The micro-service description graph belongs to the lowest abstraction layer and shows the actual behavior of the micro-service. This graph defines the internal behavior of each micro-service during the interaction in the interface description graph. The aim of this graph is to depict the behavior of the micro-service in such a way that for a SW engineer it is possible to create a totally functional piece of the SW feature. The internal behavior of the micro-service is described as a state machine. Furthermore, this graph models the behavior by using input, output, call, start, task and decision objects and the transition relationship to show the flow between the objects. This graph type does not define any service and/or interface descriptions. It just provides a link to an existing service, interface or operation, for example, using a call object. Figure 17 presents an example of the micro-service description graph.

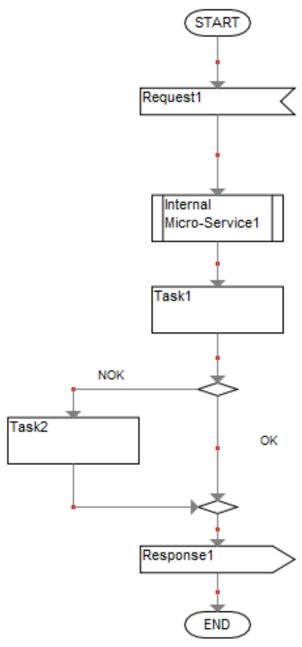


Figure 17. Micro-service description graph.

4.4. Structure of the created DSML

Based on the rules, phases and aims defined in chapter 3.2, the DSML is created for the service-based structure. The very first thing in developing DSMLs is to create a metamodel, that defines all the elements of the system, relations, restrictions, rules etc. of the system. The metamodel is based on the GOPPRR metamodeling language introduced in chapter 3.2.3. Typically, a language consists of three main components: abstract syntax, concrete syntax and semantic.

The graph tool is the most important tool of MetaEdit+ and it is used to accomplish the following tasks:

- Defining the names and properties of the graphs.
- Defining which objects, relationships and roles are used in the certain graph.
- Defining the relationships between the objects. Defining the relationships requires also defining the roles of each object in the relationship.
- Defining possible explosions and decompositions.
- Defining the constraints of each graph.

The objects and their properties are defined with the object tool, relationships and their properties with the relationship tool and roles with the role tool. Additionally, the symbol editor is used to give the finalization of the language. Defining the symbols is important to offer the easiest graphical usability to the user. Next the creation of the abstract and concrete syntaxes as well as the semantics of all the elements of the metamodel are described in detail. Henceforward, all the names of the graphs, objects and relationships are written in italics to separate them from the body text. [30]

4.4.1. Objects and Relationships of the Service Domain: Black Box

A service is a unique object that gathers micro-services into the logical entities based on the content of the micro-service. In other words, a service consists of a set of micro-services ($< \mu s >$). For example, internal communication (ICOM) area can be divided into four services (sRIO, Ethernet, Event Machine/BTS Intranet Protocol (EM/BIP) and system internal communication (SysCom)). A service includes properties such as preconditions, provided interfaces, contained micro-services and description. A service can be represented as $SRVC = \{sid, < pi >, < \mu s >, d, prc\}$, where *sid* represents a unique service, *pi* represents the provided interfaces, μs represents micro-services that belongs under a specific service, *d* represents the description of the service and *prc* represents the preconditions that the service needs to fulfill to be a functional service. Graphically it is a small rectangle inside a bigger rectangle. Figure 18 shows the graphical representation of the service and its properties.

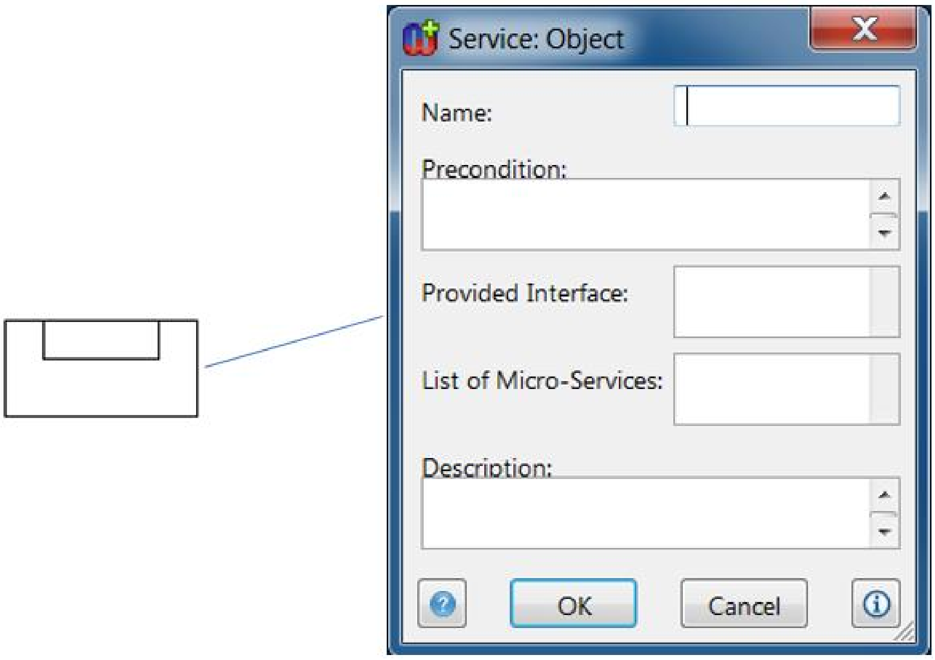
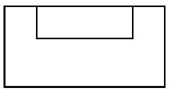
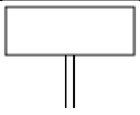
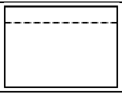


Figure 18. A service object and its properties.

The sequence diagram needs only lifeline objects that are linked to a certain micro-service and a requirement that shows the definition for the requirement where the interface description is based on. In this thesis, lifeline objects are used to represent objects that are used to model the sequence diagrams. Between lifelines, call and response message pairs communicates with the services. In the interface description graph, message pairs are defined by the call and response relationships. Both relationships have the properties of list of operations and operation sort. The list of operations property contains only one operation. Operation sort property defines whether the operation is asynchronous or synchronous.

Table 3 summarizes all the objects that belongs to the black box view metamodel. In this thesis, the data type of the property of the object is either a string, text or collection. The definitions of the data types string and text are the same as in the commonly used programming languages, but the collection as a data type is an own data type of MetaEdit+. Moreover, a collection data type needs an item type to define the collection. In this work, the item type is always an object.

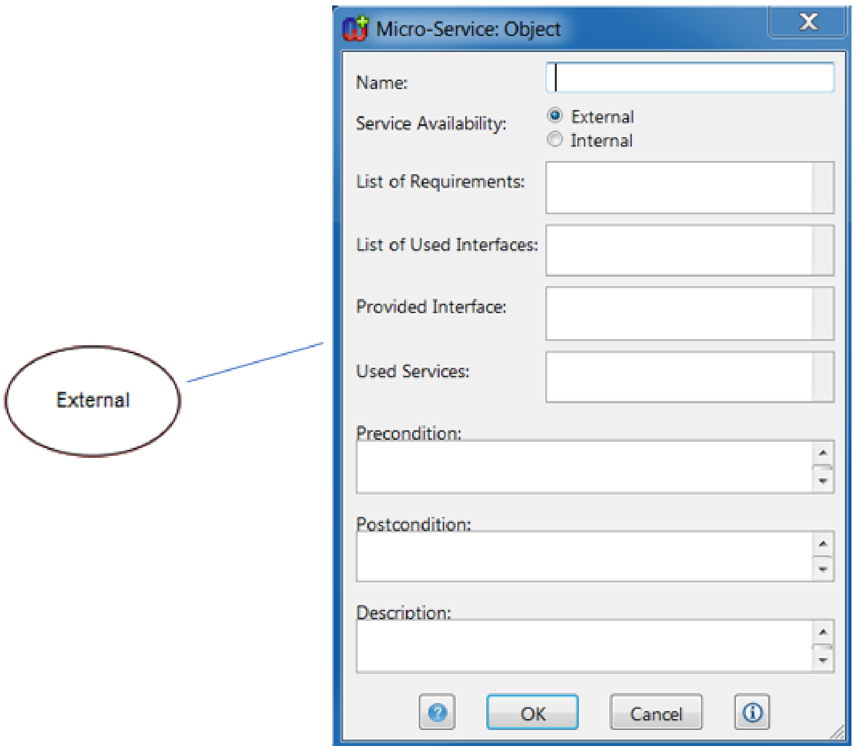
Table 3. The black box objects

Object	**Symbol**	**Property**	
		Property Name	**Data type**
Service		Name	String
		Precondition	Text
		Provided Interface	Collection: Interface
		List of Micro-services	Collection: Micro-service
		Description	Text
Lifeline		Name	String
		Interface?	Collection: Interface
		Description	Text
Requirement		Name	String
		Description	Text

4.4.2. Objects and Relationships of the Service Domain: White Box

The white box view offers the behavior actions for the service-based systems. Like the black box view, the white box view is also divided into the two parts. In this work, the first part of the white box view shows the relations between micro-services through well-defined interfaces. The second part shows a state machine like a diagram type to describe the behavior of the service itself.

A micro-service is an object that can be accessed through the service. Beside the services, the micro-services are the base of this work and they can either be internal or external. The internal micro-services are accessible only for the application whereas both the client and application have access to the external micro-services. Micro-service includes the following properties: service availability, requirements, used interfaces, provided interfaces, used micro-services, description, preconditions and postconditions. A micro-service can be represented as $\mu S = \{\mu sid, sa, r, < ui >, < pi >, u\mu s, d, prc, poc\}$, where μsid represents a unique micro-service, *sa* represents service availability, *r* represents requirement on which the functionality of the micro-service is based, *ui* represents the used interface, *uμs* represents the used micro-services, *prc* represents the preconditions and *poc* represents the postconditions. Graphically it is represented as an oval with *External* or *Internal* as a label depending on the service availability. Figure 19 shows the graphical representation of the external micro-service and its properties.



Name:

Service Availability: [✓ External] Internal

List of Requirements:

List of Used Interfaces:

Provided Interface:

Used Services:

Precondition:

Postcondition:

Description:

Figure 19. Micro-service object and its properties.

An interface is a hierarchical object that can either be provided or used depending on the relationship between the service and interface. The "main" interface can contain multiple interfaces which can also contain multiple interfaces and so on. The interface has properties, such as a list of operations, sub-interfaces and description. An interface is always linked to at least one service. The interface can be represented as $IF = \{ifid, < op >, < si >, d\}$, where *ifid* represents a unique interface, *op* is operation and *si* is sub-interface. Graphically it is represented as a circle with *IF* as a label. Figure 20 shows the graphical representation of the interface and its properties.

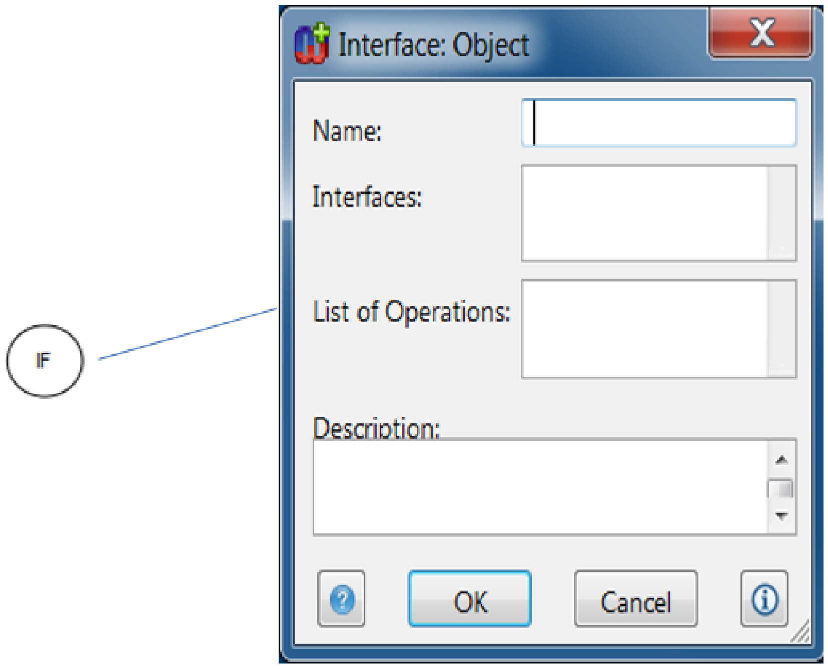


Figure 20. Interface object and its properties.

An operation includes all the messages and functions. It is an object that defines the interface. Usually messages work as request-response pairs. Parameters define the operation. An operation can be represented as $OP = \{opid, < pr >\}$, where *opid* is a unique operation and *pr* represents parameters. For example, parameters define the properties information element, priority, value range, information element type and description. Graphically operation is represented as a flash between the pointy brackets. Figure 21 shows the graphical representation of the operation and its properties.

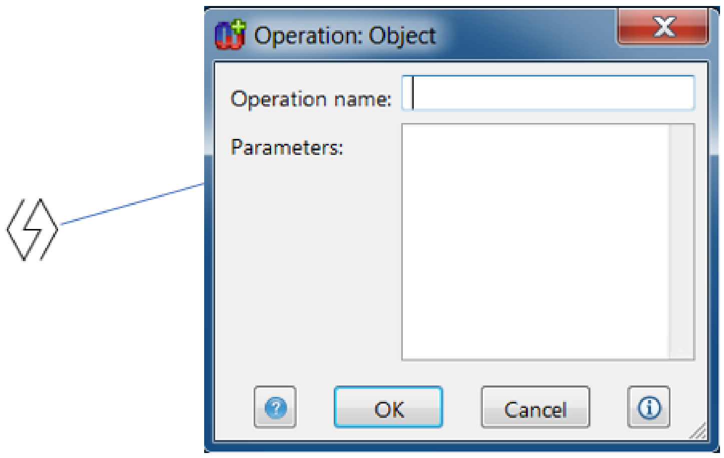
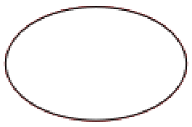
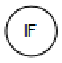
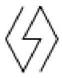
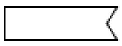
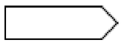
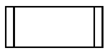
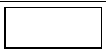


Figure 21. Operation object and its properties.

In this part of the white box view, only one relationship between objects is used. This relationship is called *connection* and it is used between micro-services through a well-defined interface. The role of the connection defines whether the interface is used or provided. A connection can have sub-graphs where the interface description of the specific micro-service and interface is defined.

The state machine like behavior uses the objects input, output, call, start, task and decision to describe the functionality of the micro-service. Both input and output are linked to a certain operation. If there are mandatory parameters, they are shown under the object. Call is linked to a micro-service that is described elsewhere in the metamodel. Call object shows, which micro-service is needed to fulfil the micro-service description. Task object shows the action that belongs only to that service that is being processed. Table 4 summarizes the white box objects and their symbols and Table 5 summarizes the white box relationships.

Table 4. Objects of the white box graphs

Object	**Symbol**	**Property**	
		Property Name	**Data type**
Micro-service		Name	String
		Service Availability	String
		List of Requirements	Collection: Requirement
		List of Used Interfaces	Collection: Interface
		Provided Interface	Collection: Interface
		Used Micro-services	Collection: Micro-service
		Precondition	Text
		Postcondition	Text
		Description	Text
Interface		Interfaces	Collection: Interface
		List of Operations	Collection: Operation
		Description	Text
Operation		Operation name	String
		List of Parameters	Collection: Parameters
Input		List of Operations	Collection: Operation
Output		List of Operations	Collection: Operation
Call		List of Micro-services	Collection: Micro-service
Task		Description	Text

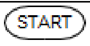
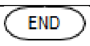
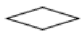
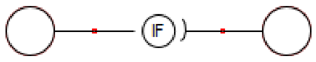
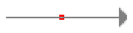
Start		
End		
Decision		

Table 5. Relationships of the white box graphs

Relationship	**Symbol**	**Property**	
		Property Name	**Data type**
Connection		Type: Provided/Used	String
Transition			

4.4.3. Rules and Restrictions

The rules and restrictions play a significant role in the modeling. As mentioned in chapter 3, by using rules and restrictions, there is a clear base for the modeling and the models are made to be functional. There are common rules and restrictions for each graph, but there are also individual rules and restrictions that are made only for a specific graph. The rules and restrictions are designed in such a way that they allow the user to model and use graphs without any problems or complications. Working with models is clear and easy. For example, naming of the graphs and elements and avoiding duplicates are more or less common restrictions when the occurrence and connectivity of the elements are strictly linked to the certain graph. The complete list of each graph type is introduced in chapter 4.5.

The constraint tool of MetaEdit+ offers four different constraint possibilities: constraint for connectivity, occurrence, port and uniqueness. The constraint for connectivity places rules for a certain object and defines how many roles or relationships of a certain kind that object may have. The constraint for occurrence defines how many times a certain object can occur in each graph. The constraint for ports defines the value of property of a certain binding of a port. The constraint for uniqueness defines which properties of a certain object need to be unique in each graph. Other rules and restrictions that cannot be defined using the constraints tool of MetaEdit+ are defined by using MERL. For example, a live check generator that constantly checks that the defined rules are fulfilled is made using MERL. The report and code generation is described in detail in chapter 4.5.5.

4.5. Service-Based Modeling with the Created DSML

This chapter contains the concrete modeling process of a feature. The previous chapter focused on the elements, rules and restrictions of the created DSML. Those concepts are used in this chapter to create a functional prototype of the SOA using the created DSML. All the elements and their properties of each graph are explained in detail as well as the report and code generators are introduced by giving concrete examples.

4.5.1. Service Group Graph (1^{st} layer)

In this thesis, the observed SW area is ICOM. ICOM can be considered as a service group. As the name says, the service group is an entity where all the services from one SW area are listed. ICOM consists of four functional services: Ethernet, sRIO, SysCom and EM/BIP. The best way to present those services is the service group graph. Therefore, the first abstraction layer consists of four service objects of the ICOM: *Ethernet* service, *sRIO* service, *SysCom* service and *EM/BIP* service of which the *sRIO* service is investigated. The service objects are named using the exact service name. Figure 22 shows the service group graph for ICOM. Basically, this graph consists only of the service objects. The graph type for the service group graphs is *BlackBoxI*. The properties of the *sRIO* service are:

- Precondition: HWAPI (Hardware Application Programming Interface) has successfully performed a start-up.
- Provided Interface: *sRIO* service provides an external *HWAPI sRIO Service* interface that includes sub-interfaces and their messages.
- List of Micro-services: *sRIO* service consists of three external micro-services: *Getting of sRIO Multicast GroupID*, *Removing Endpoint from sRIO Multicast Group* and *Adding Endpoint to sRIO Multicast Group*.
- Description states the main functionalities and aims of the *sRIO* service.

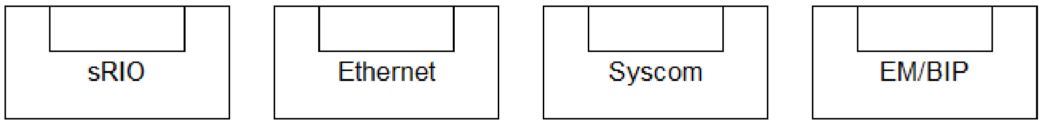


Figure 22. Overview of the service group graph for ICOM.

There exist multiple rules and conditions that are needed to consider when modeling the service group graphs. The service group graphs shall be named *Service group Name - Service_View*. In this work, the service group is ICOM, therefore the graph is named as *ICOM - Service_View*. The service group graph has constraints for uniqueness and occurrence as well as restrictions implemented using MERL. In the service group graph, the service is the only object that can directly be used, therefore all the other objects can only exist as a property of a service. Also, every service must have a unique name to avoid duplicates. All the rest of the restrictions of the properties of the service objects are done by MERL. These are:

- There cannot be used interfaces in this graph. Only provided interfaces are allowed.
- List of services must match the external micro-services in the second level. If there are differences between micro-services in the list of service

property and micro-services on the second level, the live check pane will give a warning.

Further, all of the sRIO external micro-services have a relationship to an internal micro-service. The next chapters will give a close look to all the micro-services, interfaces and all other objects that are related to the *sRIO* service.

4.5.2. Micro-Service and External Micro-Service Graphs (2nd layer)

The *sRIO* service consists of three external micro-services. The second abstraction layer graphs are used to model the micro-services and their relationships to other micro-services through an interface. All the external micro-services are linked to an individual internal micro-service. Figure 23 shows the second abstraction layer graphs (both, the *sRIO* micro-service and external *sRIO* micro-service). The graph type for the second abstraction layer graphs is *WhiteBoxI*. As the figure shows, each of the external micro-services provide the same interface, *HWAPI sRIO Service*, and use the *rio* interface that is provided by the internal micro-services. Each of the micro-service interfaces are formed with different request-response operation pairs that are modeled in the 3rd abstraction layer graphs. Next an external micro-service *Adding Endpoint to sRIO Multicast Group* is presented more closely.

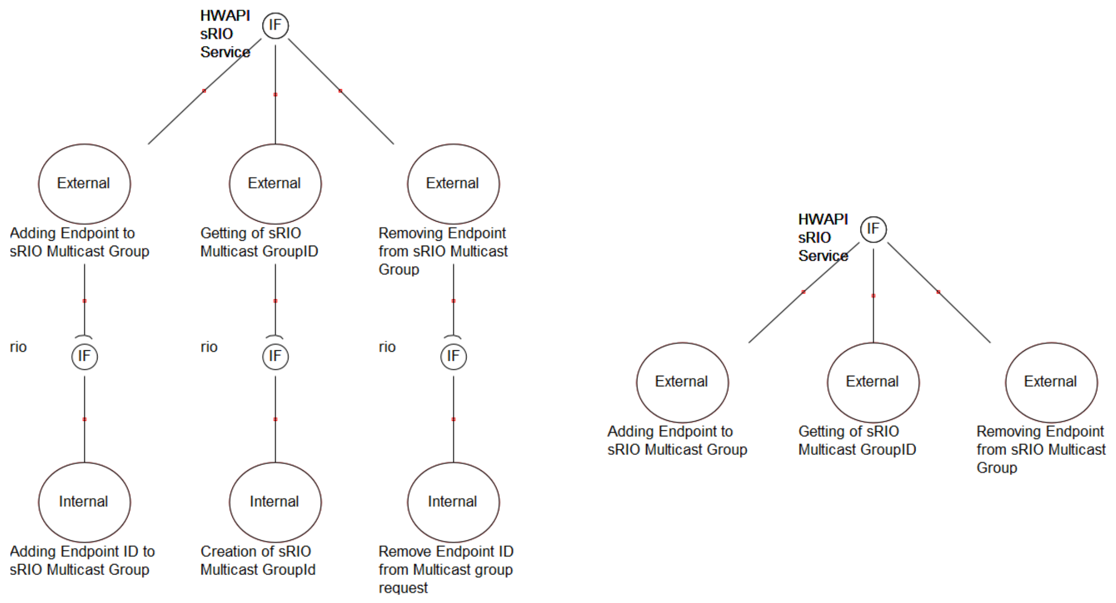
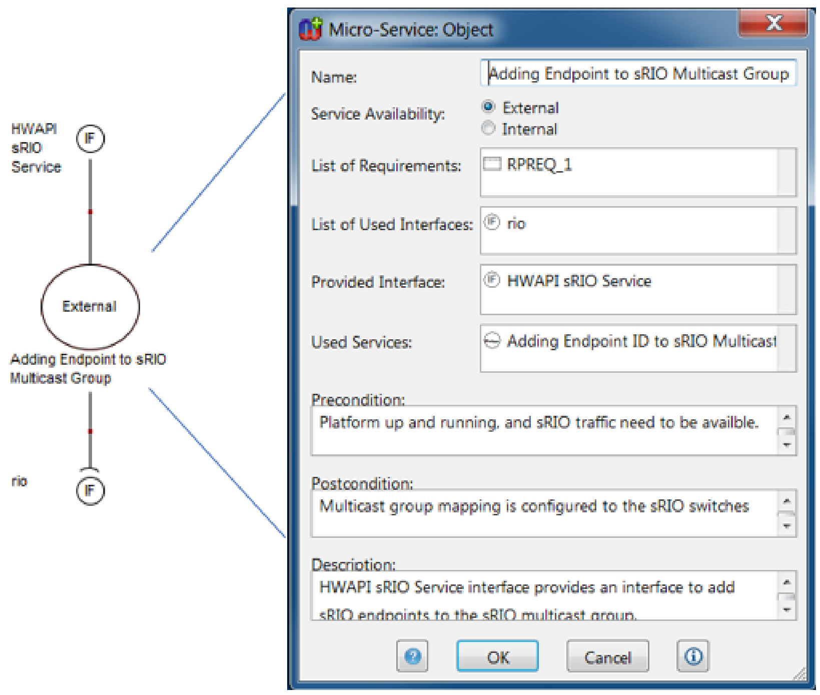

HWAPI sRIO Service
IF
External
Adding Endpoint to sRIO Multicast Group
External
Getting of sRIO Multicast GroupID
External
Removing Endpoint from sRIO Multicast Group
rio
IF
rio
IF
rio
IF
Internal
Adding Endpoint ID to sRIO Multicast Group
Internal
Creation of sRIO Multicast GroupId
Internal
Remove Endpoint ID from Multicast group request
HWAPI sRIO Service
IF
External
Adding Endpoint to sRIO Multicast Group
External
Getting of sRIO Multicast GroupID
External
Removing Endpoint from sRIO Multicast Group

Figure 23. sRIO micro-service graph on the left and sRIO external micro-service graph on the right.

Adding Endpoint to sRIO Multicast Group

Adding Endpoint to sRIO Multicast Group is an external micro-service that provides a *HWAPI sRIO Service* interface and uses an internal *rio* interface. Figure 24 shows the graphical presentation of *Adding Endpoint to sRIO Multicast Group* micro-service as well as the provided and used interfaces and its properties. Each internal and external micro-service follows the same structure that the figure depicts, therefore a one example of a micro-service is enough to present the overview of the micro-service. As properties in the figure states, the micro-service is either an external or internal and it has some pre-defined requirement. Moreover, the used and provided interfaces are shown in the micro-service properties. Preconditions and postconditions are important for the client and testing point of view. In order to execute the micro-service, the platform need to be up and running, and the sRIO traffic need to be available. To maintain the functionalities of the micro-service, the multicast mapping need to be configured to the sRIO switches. The description gives the additional information of the micro-service.



Name: Adding Endpoint to sRIO Multicast Group
Service Availability: [✓ External] Internal
List of Requirements: RPREQ_1
List of Used Interfaces: rio
Provided Interface: HWAPI sRIO Service
Used Services: Adding Endpoint ID to sRIO Multicast
Precondition: Platform up and running, and sRIO traffic need to be availble.
Postcondition: Multicast group mapping is configured to the sRIO switches
Description: HWAPI sRIO Service interface provides an interface to add sRIO endpoints to the sRIO multicast group.

Figure 24. Micro-service *Adding Endpoint to sRIO Multicast Group*.

Rules and Restrictions of the Graph

The following rules and guidelines are needed to consider when modeling the graphs from the second abstraction layer. The micro-service graphs shall be named *Service Name - Micro-service_View* and external service graphs shall be named *Service Name - External_Micro-service_View*. In this work, the service name is sRIO, therefore the micro-service graph is named as *sRIO – Micro-service_View* and the external micro-service graph is named as *sRIO – External_Micro-service_View*. The micro-service and external micro-service graph, the 2nd layer, allows only a direct use of service and interface objects. Also, the name of those objects need to be unique. To provide a better readability of the second level graphs, interface objects are reused, and therefore, if the name of the interfaces is the same in some cases, there occurs no errors, because those interfaces are not different objects. The live check generator gives errors and warnings in the following situations of the second level graphs:

- If the type of the interface object of the list of used interfaces property is ‘provide’.
- If the type of the interface object of the list of provided interfaces property is ‘used’.
- If a requirement object is missing from the list of requirements property.
- If the used service property includes forbidden services.

Also, in this graph type there are requirements for relationships. The modeling of a relationship starts always from the interface, so the relationship between a micro-service and an interface cannot start from the micro-service. As stated before, this graph type defines whether the interface is provided or used.

4.5.3. Interface Description Graph (3rd layer)

In this thesis, there are two hierarchical interfaces: *HWAPI Service* and *DDAL API* (Device Driver Abstraction Layer API) interfaces of from which *HWAPI Service* interface is an external interface and *DDAL API* interface is an internal interface. Both main interfaces consist of multiple sub-interfaces. Furthermore, each sub-interface consists of multiple operations. First, the Figure 25 shows the structure of *HWAPI Service* interface. As shown, *HWAPI Service* interface consists of *HWAPI Ethernet Service*, *HWAPI sRIO Service* and *HWAPI SysCom Service* interfaces. All the sub-interfaces consist of operations that specified the certain main interface. Thus, *HWAPI Service* interface has access to all of the operations, but the sub-interfaces have only access to the operations that specify the sub-interface itself. *DDAL API* interface follows the similar structure presented in the figure below.

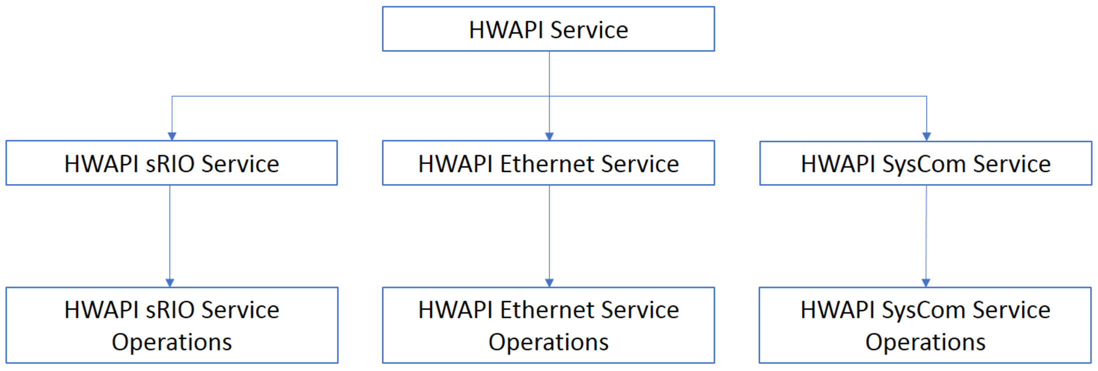
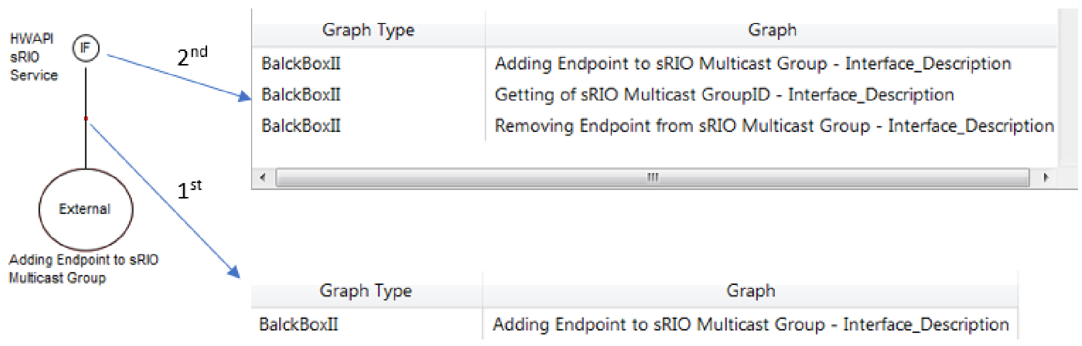

HWAPI Service
HWAPI sRIO Service
HWAPI Ethernet Service
HWAPI SysCom Service
HWAPI sRIO Service Operations
HWAPI Ethernet Service Operations
HWAPI SysCom Service Operations

Figure 25. HWAPI service interface structure.

The second abstraction layer provides access to the interface description graph(s). There are two possibilities to enter the interface description graph. The first possibility is to get access from the provide relationship role of the interface. Using this option, there is only one possible interface description graph to access. The second possibility is to get access directly from the interface. This option gives all the available micro-service description graphs that are linked to that micro-service. For example, *rio* interface is provided by three external interfaces, therefore there are three separate interface description graphs. Figure 26 shows the two possibilities to access the interface description graph(s). As the figure states, the 1st option gives only one possible graph to access and the 2nd option shows all three graphs. The graph type for the interface description graphs is *BlackBoxII*. Next, the interface description for one external and one internal service is depicted.



Graph Type	Graph
BalckBoxII	Adding Endpoint to sRIO Multicast Group - Interface_Description
BalckBoxII	Getting of sRIO Multicast GroupID - Interface_Description
BalckBoxII	Removing Endpoint from sRIO Multicast Group - Interface_Description

Graph Type	Graph
BalckBoxII	Adding Endpoint to sRIO Multicast Group - Interface_Description

Figure 26. The two possibilities to access to the interface description graph.

Adding Endpoint to sRIO Multicast Group - Interface_Description

The aim of the interface description graph is to show the interface of a specific micro-service. The interface description is modeled by using lifeline objects and request-response operation pairs. The lifeline object shows both, the interface and micro-service. *Adding Endpoint to sRIO Multicast Group* is an external micro-service that is provided by *HWAPI sRIO Service* interface. The interface description is based on the predefined requirement and the aim is *HWAPI sRIO Service* interface to add new sRIO endpoints to the sRIO multicast group. The interface is described by a request-response message pair: *API_ADD_TO_MULTICAST_GROUP_REQ_MSG* and *API_ADD_TO_MULTICAST_GROUP_RESP_MSG*. The request message has the following mandatory parameters:

- *transactionId*: an integer value that is used to associate all the messages belonging to the same procedure. All the messages that use the same procedure must use the same transaction ID.
- *endpoint*: a parameter that need to be added to the group ID.
- *groupId*: a parameter that tells the sRIO multicast group ID to where the given endpoint(s) is added.

The response message has the following mandatory parameters:

- *transactionId*
- *status*: a parameter that shows the status for the requested operation.
- *endpoint*
- *groupId*

Figure 27 shows the interface description as a sequence diagram.

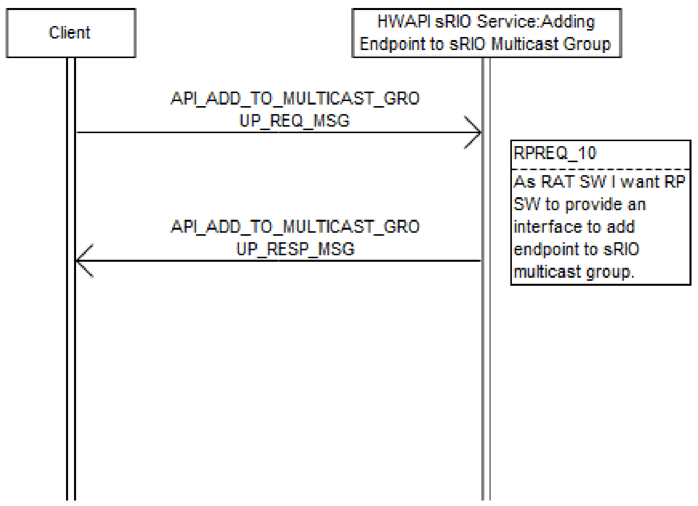

Client
HWAPI sRIO Service:Adding Endpoint to sRIO Multicast Group
API_ADD_TO_MULTICAST_GROUP_REQ_MSG
API_ADD_TO_MULTICAST_GROUP_RESP_MSG
RPREQ_10
As RAT SW I want RP SW to provide an interface to add endpoint to sRIO multicast group.

Figure 27. Interface description graph for *Adding Endpoint to sRIO Multicast Group* micro-service.

Adding Endpoint ID to sRIO Multicast Group – Interface_Description

Adding Endpoint ID to sRIO Multicast Group is an internal micro-service whose aim is to add endpoint ID to a multicast group. The interface is described by a function pair. The first function *ddal_rio_multicast_group_add* contains the parameters:

- *mcastid*: a unique integer value which is used to identify the multicast group ID.
- *deviceid*: a unique integer value which is used to identify the device ID to remove.

If the operation is successful, *DDAL_OK* function is returned. Otherwise an error code on failure is returned. Figure 28 below describes the interface description for the *Adding Endpoint ID to sRIO Multicast Group* micro-service that is provided by the *rio* interface. The function pair is modeled between the external and internal micro-services and their provided interfaces.

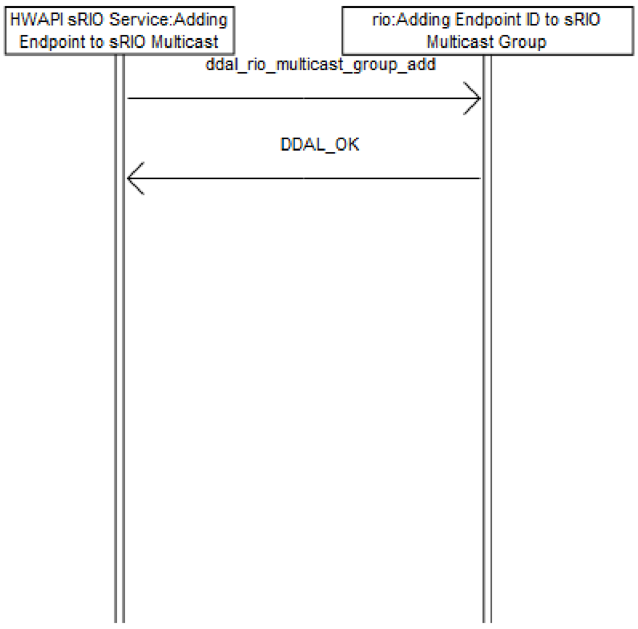

HWAPI sRIO Service:Adding Endpoint to sRIO Multicast
rio:Adding Endpoint ID to sRIO Multicast Group
ddal_rio_multicast_group_add
DDAL_OK

Figure 28. Interface description graph for *Adding Endpoint ID to sRIO Multicast Group* micro-service.

Rules and Restrictions of the Graph

The following rules and guidelines are needed to consider when modeling the interface description graphs. The interface description graphs shall be named *Micro-service Name – Interface_Description*. In the interface description graph, there can only be lifeline and requirement objects. Also, the name of both objects need to be unique. The relationship between lifelines is call or response and these relationships has an operation as a property. The name of the operation need be unique in each graph. The live check pane alerts if the requirement is missing.

4.5.4. Micro-Service Description Graph (3rd layer)

The second abstraction layer graphs provide access to the micro-service description graphs. Each micro-service has exactly one micro-service description graph where the behavior of the micro-service is described in such a way that one part of the feature can be created. Each of the micro-service description graphs start with the *start* object and ends with the *end* object. Between *start* and *end* the whole behavior of the micro-service is modeled. The graph type for the micro-service description graphs is *WhiteBoxII*.

Adding Endpoint sRIO Multicast Group – Micro-service_Description

Figure 29 describes the behavior of the *Adding Endpoint to sRIO Multicast Group* external micro-service. The behavior description starts with the *start* object and ends with the *end* object. First, an input message is shown. Also, the mandatory parameter(s) of the message is shown under the input object. In this case, *API_ADD_TO_MULTICAST_GROUP_REQ_MSG* has three mandatory parameters: *transactionId, endpoint and groupId.* Second, a call object is used to call an internal micro-service *Adding Endpoint ID to sRIO Multicast Group*. The call object does not remodel an internal micro-service, because it has already its own micro-service description graph existing elsewhere in the metamodel. After the call object, a decision is made based on the status of the operation. If the operation is successful, a task is performed where the message routing rules are configured. If the operation fails, a different task is performed where the endpoint is set to zero. Finally, an output object that is linked to a response message is modeled. The output object also shows the mandatory parameters of the operation. In this case there are four parameters: *transactionId, status, endpoint* and *groupId*. If there are no mandatory parameters, the output and input objects do not show the output pin under the input or output object. The end object states that the behavior of the micro-service has come to the end.

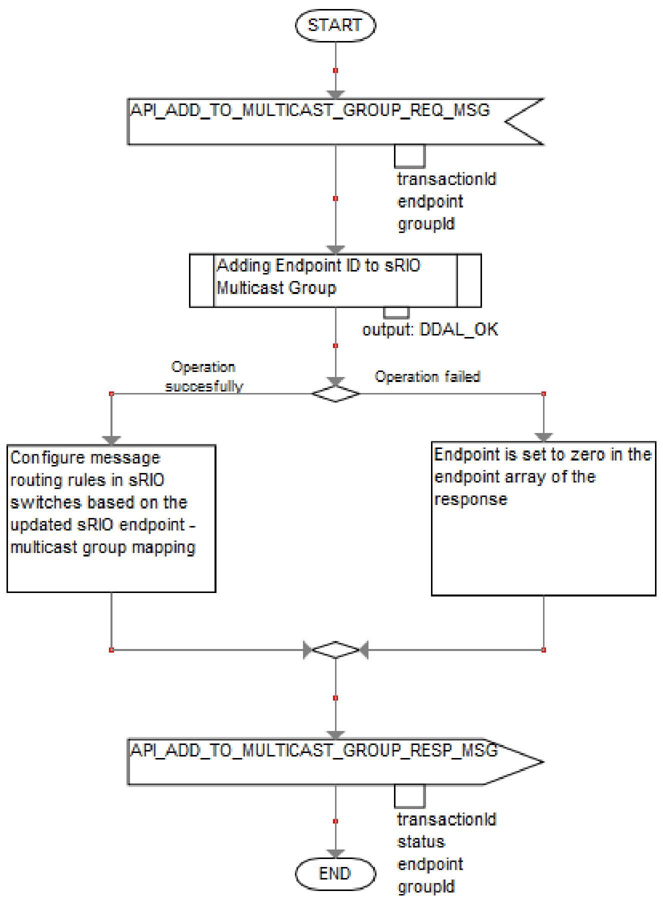

START
API_ADD_TO_MULTICAST_GROUP_REQ_MSG
transactionId
endpoint
groupId
Adding Endpoint ID to sRIO Multicast Group
output: DDAL_OK
Operation succesfully
Operation failed
Configure message routing rules in sRIO switches based on the updated sRIO endpoint - multicast group mapping
Endpoint is set to zero in the endpoint array of the response
API_ADD_TO_MULTICAST_GROUP_RESP_MSG
transactionId
status
endpoint
groupId
END

Figure 29. Micro-service description graph for *Adding Endpoint to sRIO Multicast Group* micro-service.

Adding Endpoint to ID to sRIO Multicast Group – Micro-service_Description

Figure 30 shows the micro-service description graph of the internal micro-service *Adding Endpoint ID to sRIO Multicast Group.* The structure of the graph is similar to the external micro-service description except that in the internal micro-service description graph there are no call objects used. The other observation is that the micro-service description graph is an extension of an interface description graph. Comparing Figure 28 and Figure 30, the same operations are modeled. In the micro-service description graphs, more information is modeled compared to the interface description graphs.

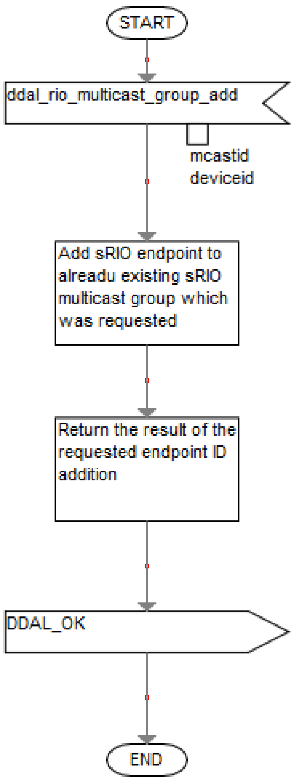

START
ddal_rio_multicast_group_add
mcastid
deviceid
Add sRIO endpoint to alreadu existing sRIO multicast group which was requested
Return the result of the requested endpoint ID addition
DDAL_OK
END

Figure 30. Micro-service description graph for *Adding Endpoint ID to sRIO Multicast Group* micro-service.

Rules and Restrictions of the Graph

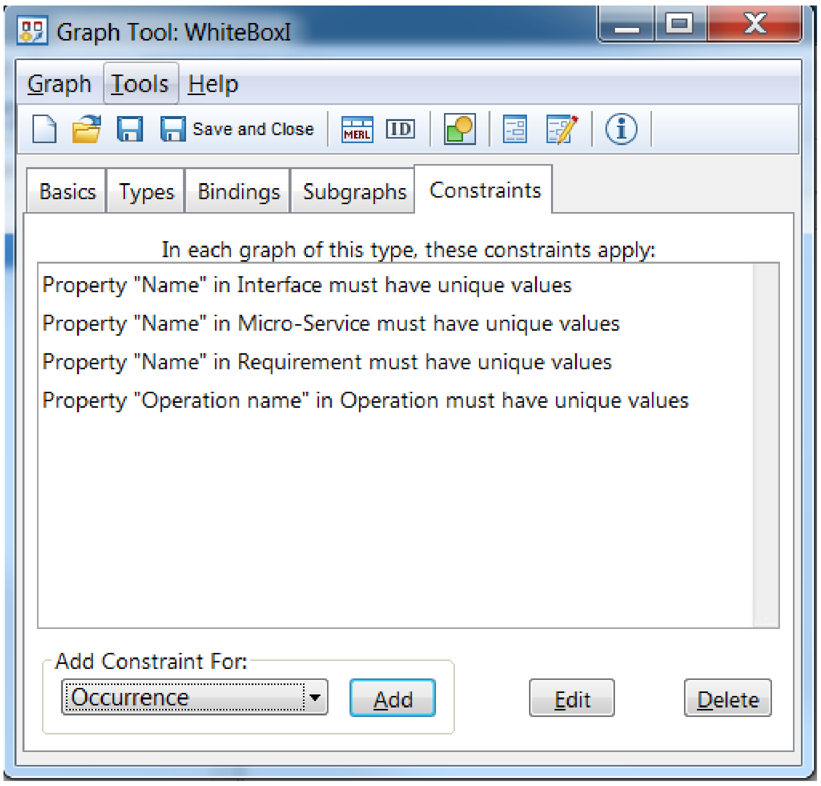
The following rules and guidelines are needed to consider when modeling the micro-service description graphs. The micro-service description graphs shall be named *Micro-service Name – Micro-service_Description.* In the micro-service description graph, there can only be white box objects. Each graph needs to start with *start* object and end with the *end* object. Moreover, each graph shall have exact one *start* and *end* object. The live check generator gives errors and warnings in the following situations:

- If the start or end object is missing.
- If the task object is empty.
- If the list of operations property of the input or output objects is empty.

4.5.5. Report and Code Generation

The code generation is the other part of the MDSD. The main focus in this work is not in the code generation. Nevertheless, basic reports and code generation are accomplished to show that the related models are certainly models, not just sketches. In this work, three kind of code generations are executed. One by using the constraint tool of MetaEdit+, one by creating a live check pane that constantly checks whether there occur errors and warnings or not, and one for the reporting that, for example, shows the objects and relationships of each graph and the most important properties of them. Basically, the reports generate the models into a text, therefore it can be seen as a model-to-text transformation. MetaEdit+ includes several built-in generators for different uses. For example, there are code generators to export graphs to HTML (Hypertext Markup Language) or word format, to generate object lists or to check some functionalities of the objects. In the following paragraphs, each three options are briefly described.

As stated before, the constraint tool provides constraints for connectivity, occurrence, ports and uniqueness. The constraints that are set by this tool are not comprehensive, but they offer the basic constraint functionality to graphs. Figure 31 shows an example of a set of constraints for the *WhiteBoxI* graphs by using the constraint tool.



In each graph of this type, these constraints apply:

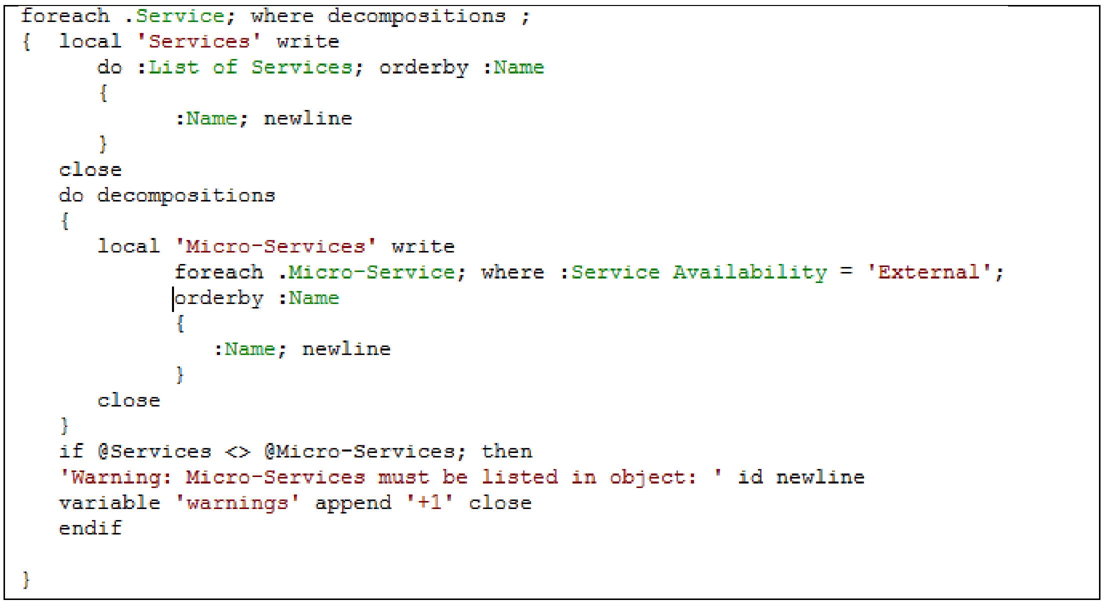
Property "Name" in Interface must have unique values
Property "Name" in Micro-Service must have unique values
Property "Name" in Requirement must have unique values
Property "Operation name" in Operation must have unique values

Add Constraint For: Occurrence

Figure 31. Constraint tool of MetaEdit+.

For example, if the modeler is trying to use a new micro-service object whose name is already used in the other micro-service in some white box graph, a pop-up window will occur and alert of a forbidden use of an object and the use of a duplicate is prevented.

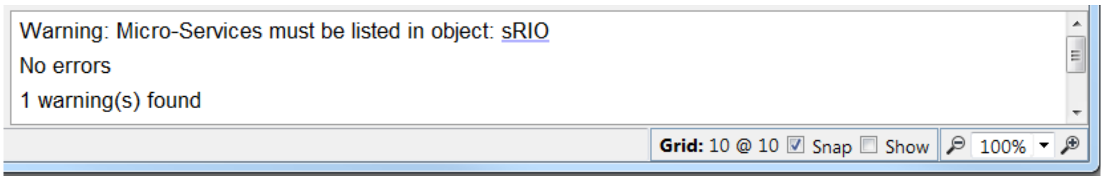
One of the main tasks of the code generator is to take care of the constraints that are not able be set by using the constraint tool. Hence, a live check pane is generated to show the real-time validity of the graphs. The live check pane reports if there occur inconsistencies and calculates the amount of the errors and warnings of each graph type. The live check pane is shown only if there is *__LiveCheck* (with two '_' as a prefix) generator in the current modeling language. Each of the graph types have their own live check generator. Figure 32 shows a piece of code that is executed with MERL. The example code checks whether the list of services property of a service matches with the external micro-services in the micro-service graph or not. Figure 33 shows the error code that is generated to the live check window pane based on the code presented below. In this case, the micro-services that are listed as a property of a service do not match with the micro-services that exist in the micro-service graph. The key of the code is that it is created to be as simple as possible. The rest of the constraints that cannot perform by using the constraint tool are executed in an equivalent way using MERL.



```
foreach .Service; where decompositions ;
{  local 'Services' write
      do :List of Services; orderby :Name
      {
            :Name; newline
      }
   close
   do decompositions
   {
      local 'Micro-Services' write
            foreach .Micro-Service; where :Service Availability = 'External';
            orderby :Name
            {
               :Name; newline
            }
      close
   }
   if @Services <> @Micro-Services; then
   'Warning: Micro-Services must be listed in object: ' id newline
   variable 'warnings' append '+1' close
   endif

}
```

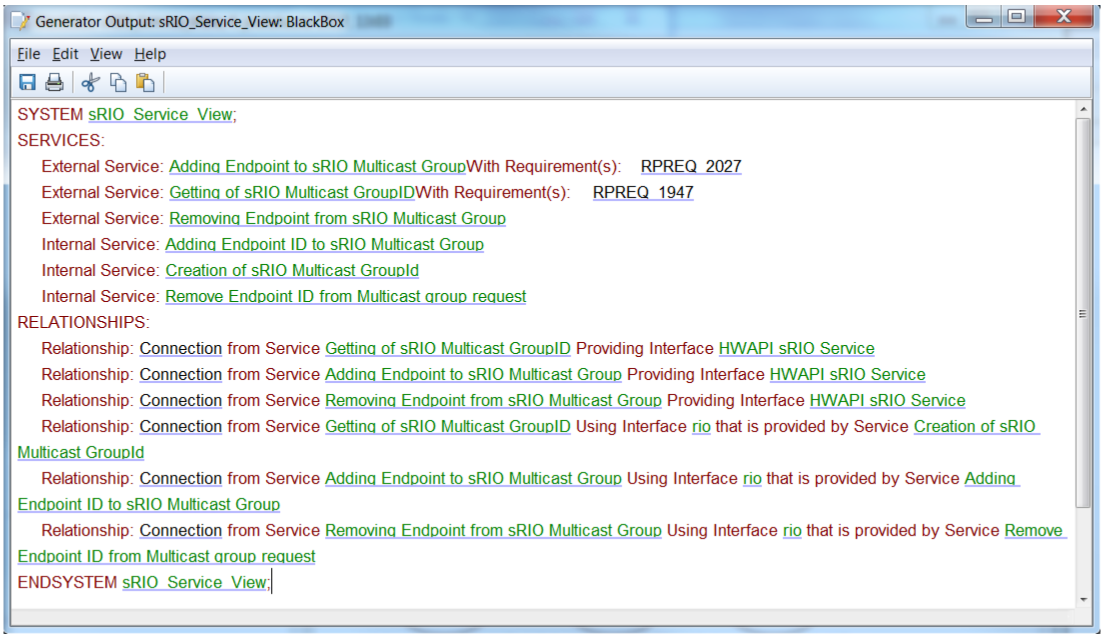
Figure 32. An example of MERL.



```
Warning: Micro-Services must be listed in object: sRIO
No errors
1 warning(s) found
```

Figure 33. Live check window pane.

Generating reports is the third option to use code generators. Reports transform the models to a text format that supports the readability of the models. Figure 34 shows the output of the generated report of the *sRIO_Service_View* graph (Figure 23). The report shows all the services, both the internal and external interfaces and the relationships between them. From the report, the highlighted hyperlink of the object, graph or relationship can be chosen and the hyperlink shows the actual place of the element in the graph. The hyperlink can be chosen and the properties of the element can be accessed. Thus, reports offer a traceability among elements and graphs. The reports show the specified characteristics for each graph type. Therefore, the generated output of the reports for each graph differs, but the main principle of generating the reports is the same.



```
SYSTEM sRIO_Service_View;
SERVICES:
    External Service: Adding Endpoint to sRIO Multicast GroupWith Requirement(s):   RPREQ_2027
    External Service: Getting of sRIO Multicast GroupIDWith Requirement(s):   RPREQ_1947
    External Service: Removing Endpoint from sRIO Multicast Group
    Internal Service: Adding Endpoint ID to sRIO Multicast Group
    Internal Service: Creation of sRIO Multicast GroupId
    Internal Service: Remove Endpoint ID from Multicast group request
RELATIONSHIPS:
    Relationship: Connection from Service Getting of sRIO Multicast GroupID Providing Interface HWAPI sRIO Service
    Relationship: Connection from Service Adding Endpoint to sRIO Multicast Group Providing Interface HWAPI sRIO Service
    Relationship: Connection from Service Removing Endpoint from sRIO Multicast Group Providing Interface HWAPI sRIO Service
    Relationship: Connection from Service Getting of sRIO Multicast GroupID Using Interface rio that is provided by Service Creation of sRIO
Multicast GroupId
    Relationship: Connection from Service Adding Endpoint to sRIO Multicast Group Using Interface rio that is provided by Service Adding
Endpoint ID to sRIO Multicast Group
    Relationship: Connection from Service Removing Endpoint from sRIO Multicast Group Using Interface rio that is provided by Service Remove
Endpoint ID from Multicast group request
ENDSYSTEM sRIO_Service_View;
```

Figure 34. Output of the generated report.

5. DISCUSSION

The aim of this thesis was first to define a prototype of a SOA and second to create a prototype of DSML, and use the created DSML to build the proposed SOA. Chapter 4.3 depicts in detail all the abstraction layers and the graphs of each layer that are used to model a SW feature resulting in an executable service-based architecture. The requirements for evaluating the proposed SOA are presented in chapter 2.2.3. All the graphs are reviewed by the SW specialists and the discussion whether the proposed SOA fulfils the requirements or not is based on the reviews and the user experience. Also, the created DSML is discussed from a modeling point of view based on the feedback from the SW specialists and theory. Finally, a short tool evaluation is given.

5.1. Discussion of SOA

In this chapter, the proposed SOA is evaluated based on the reviews of the SW specialists and the requirements that were listed in chapter 2.2.3. Based on the evaluation of the requirements, the hypothesis presented in chapter 4 is validated.

The proposed SOA and the metamodel was introduced to the SW specialists and the review comments were gathered. The SW specialists have experience on the component-based modeling, and therefore they could make straight comparisons between the service-based and the component-based modeling. The review comments can be divided into two parts:

1. Service-based modeling from the specification perspective.
2. Service-based modeling from the reader and/or service user perspective.

From the SW specialist perspective, the service-based modeling provides automatically clear and solid structure for the specification, which helps to find the impact of the new features to the specification, to make new specifications and to maintain the architecture, interfaces and the design. Moreover, the service-based approach states making the specifications to the direction where the specifications are made for the services instead for components. The current way of modeling is being driven to a similar direction, but still combining services and components, which makes the specifications somewhat complex to read.

From the reader of the specification perspective, the service-based approach provides a solid structure which helps the reader to find the services and the needed specification. Compared to the current component-based approach, the service-based approach provides much more extra value to the reader of the specification. For example, comparing the interface definitions, the service-based approach offers better visibility to the reader than in the current modeling. The current component-based structure does not offer a black box view for the client because the structure is done only for the modeler purposes. The proposed SOA offers different views for the client and modeler which broadens the use of the models.

The comments from the SW specialists favor the proposed service-based approach over the current component-based approach. According to the comments, the service-based approach offers more logical structure by minimalizing the number of the abstraction layers. However, the proposed SOA is just a tiny part of the working SW specification structure. The next step is to expand the definition and usage of SOA to give a better understanding of the service-based modeling. Based on the review comments of the SW specialists, a service-based approach in the scope of this work is a functional product and it is definitely worth further investigation.

5.1.1. Requirements

The requirements are evaluated based on the levels of fulfilment that are [36]: fulfilled, partly fulfilled and poorly fulfilled depending on how well the proposed SOA prototype fulfils the requirements.

REQ1 considered the loose coupling. Each element of the SOA should be autonomous and the relationships between the elements are defined to ensure system consistency. In the proposed SOA, each element has an important and a well-defined role. Each element has an important role and the proposed SOA cannot be executed if one of the element is missing or malfunctional. The communication between the services and the client is compiled with operations. REQ1 is the base for the proposed SOA prototype. The aim of the related prototype is to consist of well-defined independent elements that have relations trough an interface. Thus, each element is an independent element that has a certain well-defined purpose. Based on the definition of REQ1, it could be stated that the fulfilment level of this requirement is fulfilled.

REQ2 considered the autonomy of the services. Each service can control itself without dealing with many external dependencies to other services. In the related prototype, there are dependencies between micro-services through an interface. However, each service and micro-service is implemented individually and they have their own service and interface descriptions. Based on the definition, REQ2 could be stated as partly fulfilled.

REQ3 considered the abstraction of SOA. The aim is to hide the implementation details using abstractions and the interfaces are used as access points. This requirement is highly linked to the created modeling language. The proposed SOA provides three different abstraction layers which all have their own aim. Each layer has certain graph types that have a certain set of objects. For example, the service objects act like black boxes hiding the information and the logic that are described in the lower abstraction layers. Moreover, one of the core idea behind the proposed SOA was that there are different views. In the proposed SOA, there are black box and white box views. Black box views for client's purposes and white box views for SW specialist's purposes. Based on this, REQ3 could be stated as fulfilled.

REQ4 considered the reusability of the elements of the SOA. The elements, especially the services and micro-services, should be designed as independent as possible. In the proposed prototype, all the services and micro-services are somewhat specific, thus they cannot be designed as fully generic services. Nevertheless, micro-services can be called from another graph to avoid duplicates. In addition, other elements such as operations and interfaces are implemented by means of being generic. Each element is designed to be reusable. Based on this, REQ4 could be stated as fulfilled.

REQ5 considered the discoverability of the services. The SOA needs to provide the discoverability of the services which is commonly implemented by a service repository. Compared to other requirements, REQ5 is the most tool-based requirement. MetaEdit+ is a repository based tool that provides that option. To be discoverable, the redundant services need to be avoided. The proposed prototype alerts if the name of the service is already taken, but there are no other mechanisms to avoid redundant services. Even if there exist redundant elements, repository should be defined in a such a way that finding the needed elements is easy. MetaEdit+ offers solid searching possibilities. Thus, REQ5 could be stated as partly fulfilled.

REQ6 considered the coarse-grained services. There are various levels of granularity, and finding the right granularity for the observed services is a challenge. Fine grained services exchange only small amounts of data and are more limited, whereas coarse-grained services are able to exchange more data, but encapsulate more functions within a service request. In the proposed porotype, service objects can be considered as coarse-grained services and micro-service objects as fine-grained services. Thus, REQ6 could be stated as partly fulfilled.

The final requirement, REQ7, considered the interface-based design of the SOA. Also, REQ3 and REQ4 partly touched the interface-based design. In the proposed prototype, each service and micro-service will interact with another service or micro-service through a well-defined interface. Hence, the interface is one of the key elements of the proposed metamodel. The one way to observe interface-based design is that the interfaces will hide the service and micro-service. In the proposed SOA, interface has its own properties and descriptions. Thus, REQ7 could be stated as fulfilled.

To summarize, each of the requirements were either fulfilled or partly fulfilled. Moreover, only REQ2, REQ5 and REQ6 were partly fulfilled. Based on the level of fulfilment of the requirements, the proposed porotype of the SOA could be considered successful.

5.1.2. Hypothesis

As stated before, the current component-based modeling approach is problematic due to the construct of the large-scale enterprise and the responsibility areas of each SW teams. As a reminder, the hypothesis that this thesis tries to address is:

- H1: A service-based approach might solve the problem that the current component-based approach has faced.

The hypothesis is validated either as true or false depending on whether the requirements stated above are fulfilled or not. As stated in the end of chapter 5.1.1, the SOA prototype was successful. When it comes to the requirements, the hypothesis is validated. From the seven requirements, four were fulfilled while three were partly fulfilled resulting to the conclusion that the hypothesis is true. Therefore, the proposed metamodel fulfils the requirements to be a service-based model. Also, based on the review comments, the proposed SOA is a functional product. Thus, there are open questions relating to the hypothesis. The hypothesis states that the SW components are becoming too extensive to be efficient by means of modeling, because the construct of the components are becoming illogical. Thus, the main key performance indicators for both service-based and component-based approaches are the size of the scope per service and the possibility to create functional entities simultaneously in a faster and efficient way. But what is the right size? Thus, the question is, can the SW be decomposed into a set of services which have the “right” size? The prototype presented in this work cannot give a clear answer because the service set is limited. For the future work, the service-based approach presented in this work needs to be taken to a broader use to see whether the right size of the service can be defined in such a way that the modeling work is more efficient compared to the current component-based structure.

5.2. Discussion of DSML

Before the beginning of the work, the domain was already chosen. The biggest problem was to limit the domain area in such a way that the work would not expand too much, but still keep the content of the domain vast enough to be able to model all the needed characteristics of the problem domain. When developing DSML, both domain and language development expertise are needed. The author of this thesis did not have previous experience on any kind of language design, service-oriented architecture or code generator, and therefore everything was started from a clean sheet. As a result, a prototype of a service-based architecture using DSML was successfully created.

The evaluation of the modeling language was based on the user experience and the theory behind the DSML creation. The requirements for the modeling language were asked from the SW specialists that use modeling languages in day-to-day use. The requirements of the created DSML are evaluated with the same scale of level of fulfilment as the requirements for SOA: fulfilled, partly fulfilled and poorly fulfilled, based on how well the proposed DSML fulfils the requirement. The following list of requirements for the modeling language was the outcome:

1. Language REQ1: The created DSML should be graphically good.
2. Language REQ2: Each element of the created DSML should be suitable named.
3. Language REQ3: Each element of the created DSML should have well-defined constraints.
4. Language REQ4: The created DSML should be easy to use for the modeler no matter the modeling experience level.
5. Language REQ5: The created DSML should be easy to modify.
6. Language REQ6: The domain of the created DSML should be well-defined.
7. Language REQ7: The created DSML should be composed of multiple layers.

DSML should easily be presented via its graphical presentation. The language REQ1 discussed the graphical presentation of the proposed modeling language. In this thesis, each object and relationship has their own symbol and icon representations that cannot be mixed up. The aim was to create as simply graphical representation for each object as possible. Some of the symbols were learnt from the available examples, but most of the symbols were the own creation of the writer of this thesis. As a result, the graphical notation of the objects and relationships were successfully created. The language REQ1 can be stated as fulfilled.

The naming of the objects, graphs and relationships need to be done in such a way that the usage of the modeling element is clear. The language REQ2 discussed the naming of the elements of the proposed DSML. The naming of the elements was clear because all the elements had their own usage. In this thesis, there were no lookalike elements. Each element had their own specific usage and area of responsibility. For example, each of the micro-services were named based on the main functionality of the micro-service. Therefore, one can see the use of the service directly from the naming of the object. There is no need to go deeper inside the objects. Names of the operations and the interfaces were pre-defined; therefore, the naming of those objects was clear. The language REQ2 can be stated as fulfilled.

The language REQ3 discussed the constraint of the modeling language. Each graph, object and relationship need to have well-defined constraints to provide a functional model. The constraints are discussed in detail in chapter 4. The aim of this thesis was to find a way to define models based on the service-based approach, and the language creation was performed after the SOA was defined. The constraints of the modeling language are already defined in the different modeling environment, therefore most of the constraints could be copied to MetaEdit+ environment. Hence, these modeling environments differ from each other, and therefore lots of different constraints compared to the existing ones needed to be defined. The language REQ3 can be stated as partly fulfilled.

The language REQ4 discussed the usability of the language among different expertise levels of the language developer. As stated before, the writer of this thesis did not have any previous experiences of DSMLs. Thus, the usage of the language is defined to be suitable for the DSML experts as well as the novices. Therefore, the language REQ4 can be stated as fulfilled.

The language REQ5 discussed the ability to modify the DSML. This requirement is highly related to the modeling tool. In chapter 3.2.3, the main functionalities of MetaEdit+ modeling tool were discussed. MetaEdit+ consists of the workbench and modeler licenses where the modeling language is defined using workbench license and the defined modeling language is used by using modeler license. Therefore, using the modeler license, the modifications of the DSMLs are restricted. Thus, from the modeling point of view, all the properties that are allowed to modify using the modeler license is made easy. For example, the proposed DSML offers different menus where certain objects can be chosen based on the constraints and restriction of the language defined with the workbench license. As a result, the language REQ5 can be stated as fulfilled.

The investigated domain was already decided and defined before the work started. To be a well-defined domain, all the needed elements need to be properly defined to allow efficient way of modeling. The elements have been investigated thoroughly and the constrains and the rules have been defined to fulfill the requirements of each element. Therefore, the language REQ6 can be stated as fulfilled.

The final language requirement, language REQ7, discussed the construction of the proposed DSML. This requirement is similar to the SOA requirement REQ3 where the abstraction layers of SOA were discussed. Similarly, the proposed DSML is used in the three different abstraction layers. Each of the layers have their own graph types that are defined by certain objects. Moreover, in the proposed DSML, there exists hierarchical elements to provide the abstraction of the language. For example, interface objects consist of operation objects which consist of parameter objects. Therefore, the language REQ7 can be stated as fulfilled.

As a result, the language REQ3 was the only language requirement that was not fully fulfilled. However, the language REQ3 was stated as partly fulfilled. Based on the level of fulfilment, it can easily be said that the proposed DSML is designed and developed successfully within the scope of this thesis. Though, because the aim of this thesis was to concentrate more on the new service-based modeling approach than on the creation of the language, only the needed structures of the language were created. The structure of those objects was made as simply as possible to demonstrate successfully the functionality of the service-based modeling approach.

5.3. Tool Evaluation

Even though the author of this thesis had a limited knowledge of the modeling tools, MetaEdit+ tool was quite easy to learn. Defining DSML with MetaEdit+ tool was somewhat slow due to the nature of DSML creation. Hence, every element and property of the language needed to be created before it could be used, and therefore, at the beginning of the modeling work there were no existing elements. Thus, this way of defining the language provides exactly the kind of language that the creator of the language wants.

The biggest benefit of the tool was that the language developer can develop exactly that kind of language that the developer wants to. There are infinite number of possible ways to implement a new DSML. Of course, there are examples of existing DSMLs therefore there is no reason to invent everything again. For example, MetaEdit+ provides its own demo repository where different DSML examples can be found.

When all the basic concepts of using MetaEdit+ were studied and learned, the language creation was efficient and dynamic. Each element could be developed using a specific tool for that element. For example, defining objects, an object tool is used and to define relationships, a relationship tool is used. The most time-consuming part of the language creation in this thesis was developing the code generators. The manuals for the MERL were good but the learning of the MERL comes from coding the language itself. Developing a DSML and code generators is an iterative process. Thus, a tiny part of the model and a code generator for that at the time need to be created and tested. If the name of some object, relationship or property was changed, also the code generators need to be modified. Therefore, changing the names of DSML elements need to be avoided. If there were problems in the DSML or code generation creations, the support of MetaCase was always available. Therefore, before starting the concrete DSML developing work, the background and the aims of the DSML should be clear. This eases the work and significantly reduces the time of developing the DSML.

Of course, there does not exist a prefect modeling tool. Each of the available tools have their own weaknesses. In the presented metamodel, the biggest aim of the development from the tool point of view would be the ability to find specific information of a specific element. If a specific parameter of a specific operation that is linked to a specific micro-service wanted to be checked quickly, a lot of clicking and windows need to be opened. Generally, the navigation between models and elements were sometimes troublesome. This might be because of the lack of experience of creating the DSML of the author of this thesis, or just a tool implementation problem. Moreover, there were found similar cases where a simple information of the model needed to be found and lots of windows needed to be opened to find an information source. However, the information was existing, it just needed a bit more effort to be found. Using code generators more efficiently, the needed information could be provided to be more visible. Also, the constraint tool was a bit concise. For example, there is no possibility to restrict that in the graph there can occur one and only one object of a certain type. For example, start and end objects needed to have such a restriction. Now, this restriction needed to be made using MERL. Therefore, in the future, there could be a choice to add constraint to objects of a certain type that may occur exactly one time in each graph.

In the end, MetaEdit+ is a tool for creating DSMLs and each of the DSMLs are different. Hence, it would be impossible to create a tool that will fulfil the needs of every SW developers or enterprises perfectly. Thus, except the minor issues of the tool, MetaEdit+ is a suitable tool for creating the DSML at least in the small to medium size complex SW systems.

6. SUMMARY

The increased need for software and functionalities behind the software has made the system design work more and more demanding and the components of the current component-based structure are becoming too large to be able to work efficiently. The component-based structure is not any longer the perfect option to create functional entities simultaneously in a fast and efficient way. The main aim of this thesis was to find an alternative for a component-based SW modeling. The service-based modeling approach was a potential choice to replace the component-based approach. DSML was used to utilize the creation of the service-based modeling. An existing SW feature, which was implemented by using the component-based modeling approach, was remodeled from the SOA point of view. The hypothesis of the thesis was that a service-based approach might solve the problems that the current component-based approach has faced.

Software technology has evolved over the years. At the beginning of this thesis this evolution is introduced from object-oriented SW to service-oriented SW. The focus of this thesis was in the SW system development. The SW system discussed in this thesis was BTS platform SW and its functionality was briefly explained by giving theory and examples. The common SW development methods were also introduced. Waterfall method is the base of the different SW development methods. More up-to-date SW development methods such as spiral model, prototyping and iterative and incremental development are evolved from the waterfall method. The last introduced SW development method in this thesis was agile and its different variations such as scrum, XP, FDD and TDD. Moreover, different SW testing strategies and MBT were briefly discussed.

MDSD is one of the key aspects on which this thesis relies. There are several ways how MDSD can be realized. In this thesis, MDA and DSM are presented to support MDSD. MDA approach introduces the metamodeling and different metalevels. In addition, levels of abstractions are introduced and the differences between diagrams and models are explained to avoid misunderstanding.

DSM has a couple of main aims. First, to raise the level of abstraction using a modeling language that is created to solve a problem using concepts and rules. Second, to develop the final product by using a chosen programming language or other form from the used specifications. Third, to enable code generation. DSM and DSML are used for modeling the SOA. The DSML development process and designing guidelines to define the DSML are introduced. Moreover, the DSM tool MetaEdit+ is used to create the DSML. In this thesis, an overview of MetaEdit+ and its main properties is given.

In the practical part of this thesis, a feature of BTS platform SW was modeled using the created DSML and proposed service-based modeling approach. The feature was taken from ICOM area and the feature was related to defining multicast operations of sRIO. The feature was modeled based on the model elements and graphs defined via the created DSML. The aim was to remove the SW components and replace them with the services resulting in a more efficient way of modeling features. The SOA and all the elements of the SOA need to be defined. First, the different abstraction layers of the service domain and the graph types that belongs to the specific layer were described. Then, the SOA metamodel was divided into two parts based on the usage of the objects: black box and white box. Further, both were again divided into two parts because of the properties of the modeling tool. Black box parts contain objects service, lifeline and requirement and the first part of the white box part contains objects micro-service, interface and operation. The black box elements are used to provide an overview of the feature without showing the internal functionalities of the services. The second part of the white box contains all the objects that models the behavior of the micro-service. These objects were start, end, input, output, decision, call and task. The white box elements are used to describe the internal functionality of the services.

The proposed SOA prototype consists of three abstraction layers. The first layer contains the service group graphs. The second layer contains both the external micro-service graphs and the micro-service graphs. The lowest layer contains the interface description graphs and the micro-service description graphs. The service group graphs and interface description graphs are considered black boxes and other graphs white boxes. After the graphs and objects of each graph were defined, all the constraints and restrictions were defined to prevent the misuse of the elements. MetaEdit+ offers two possibilities to create constraints. One is by using the constraint tool of MetaEdit+ and the other is to use code generators to define live check generator that shows the errors and warnings in real time when modeling.

After all the elements of the SOA prototype were defined and created, the concrete modeling could be started. All the abstraction layers were modeled using the created DSML. After all the graphs were modeled, a report and code generators were defined to denote the functionality of the models. Using code generators, the difference between models and drawings can be validated. The code generators were defined using MERL. MERL is an own object-oriented based scripting language of MetaEdit+. Each of the graph type had their own generated reports. The output of the generated reports showed, for example, the used objects and their main properties as well as the relationships between different objects.

Based on the requirements of the SOA and the review comments of the SW specialists, the service-based modeling approach was evaluated. The evaluation was based on the levels of fulfilment that are: fulfilled, partly fulfilled and poorly fulfilled. Four of the seven given requirements were fulfilled while three of the seven requirements were partly fulfilled. Based on the requirements and the review comments, the hypothesis could be validated as true. However, some questions raised up based on the validation of the hypothesis. The hypothesis states that the SW components are becoming too extensive to be efficient by means of modeling, because the construct of the components are becoming illogical. Thus, the main key performance indicators for both service-based and component-based approaches are the size of the scope per service and the possibility to create functional entities simultaneously in a faster and efficient way. But what is the right size? Thus, the question is, can the SW be decomposed into a set of services which have the “right” size? The future work should take the proposed service-based model into broader use.

Based on the reviews and user experience of the SW specialists, the created DSML was evaluated. The same grade of fulfilment was used to evaluate the created DSML. The language REQ3 was the only language requirement that was not fulfilled. However, the language REQ3 was stated as partly fulfilled. As a result of the evaluation, the proposed DSML was created successfully. However, because of the scope of this thesis, the focus was more in the service-based modeling than in the DSML creation. Thus, only the needed characteristics of the DSML were defined.

Finally, a modeling tool MetaEdit+ was valuated. Despite the minor problems that occurred during the thesis work, MetaEdit+ proved to be a solid tool for creating service-based models using DSML.

7. REFERENCES

[1] Brown A., Johnston S. & Kelly K. (2002) Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Rational Software Corporation.

[2] Kern H. (2008) The Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges. In 8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA 2008. University of Alabama at Birmingham, pp. 14-19.

[3] BTS System Reference Document (Read 1.1.2018) v2.0. URL: http://www.obsai.com/specs/OBSAI_System_Spec_V2.0.pdf

[4] Ruohonen N. (2017) System Modeling to Enable Model-Based Testing in Large-Scale IT-Business. Master's Thesis. University of Oulu, Department in Electrical Engineering, Oulu.

[5] Seth A., Aggarwal H. & Singla A.R. (2011). Evolution of technology through procedural, object oriented, component based to service oriented. Journal for Computing Teachers.

[6] Lethbridge T. & Laganiere R. (2001) Object-Oriented Software Engineering: Practical Software Development Using UML and Java, 2nd ed. McGraw-Hill education, Berkshire, 533 p.

[7] Lee R. (2013) Software Engineering: A Hands-on Approach. Atlantis Press, 288 p.

[8] Goma H. (2011) Software Modeling and Design. Cambridge University Press, 592 p.

[9] Swain G. (2010) Object-Oriented Analysis and Design Through Unified Modeling Language. University Science Press. An Imprint of Laxmi Publications, 236 p.

[10] Zhang Z. (2001) Components Analysis in the Metamodelling Based Information System Development. University of Jyväskylä, Department of Computer Science and Information Systems, Jyväskylä.

[11] OASIS (read 10.1.2018). Reference Model for Service Oriented Architecture 1.0. URL: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

[12] Bean J. (2010) SOA and Web services interface design - principles, techniques, and standards. Morgan Kaufmann/Elsevier, 384 p.

[13] Sterff A. (2006) Analysis of Service-Oriented Architectures from a business and an IT perspective. Master's Thesis. Technical University of Munich, Department of Informatics, Munich.

[14] Beizer B. (1995) Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, 294 p.

[15] Jacob P.M. & Prasanna M. A (2016) Comparative analysis on Black box testing strategies. International Conference on Information Science (ICIS), Kochi, pp. 1-6.

[16] Utting M. & Legeard B. (2006) Practical Model Based Testing: A Tools Approach, Morgan Kaumann 1st ed., 456p.

[17] Schultze C., Lindvall M., Bjorgvinsson S. & Wiegand R. (2015) Model generation to support model-based testing applied on the NASA DAT Web-application – An experience report. In: IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, Maryland, pp. 77-87.

[18] Merilinna J., Puolitaival O-P. & Pärssinen J. (2008) Towards Model-Based Testing of Domain-Specific Modelling Languages. 8th OOPSLA Workshop on Domain-Specific Modeling Languages, Nashville, Tennesee.

[19] Merilinna J. & Puolitaival O-P (2009). Using Model-based Testing for Testing Application Models in the Context of Domain-Specific Modelling. 9th OOPSLA Workshop on Domain-Specific Modeling, Orlando, Florida.

[20] Puolitaival O-P. & Kanstrén T. (2010) Towards Flexible and Efficient Model-Based Testing, Utilizing Domain-Specific Modelling. In proceedings of the 10th Workshop on Domain-Specific Modeling, Reno, Nevada.

[21] IRMA (2014). Software Design and Development: Concepts, Methodologies, Tolls, and Applications, Volume 1. IGI Global.

[22] Stober T. & Hansmann U. (2010) Agile Software Development: Best Practices for Large Software Development Projects. Springer, 179 p.

[23] Cho H. (2013) Domain-Specific Modeling Language Creation. A Dissertation. The University of Alabama, the Department of Computer Science, Alabama.

[24] Larman G. & Brasili V. (2003) Iterative and Incremental Development: A Brief History. IEEE Computer, vol. 36, no. 6, pp. 47-56.

[25] Boehm B. (1988) A Spiral Model of Software Development and Enhancement. IEEE Computer, IEEE, pp. 61-72.

[26] Lichter H., Schneider-Hufschmidt M. & Zullighoven H. (1993) Prototyping in industrial software projects-bridging the gap between theory and practise. In proceedings of the 15th International Conference in Software Engineering. IEEE, Baltimore, MD.

[27] Stahl T. & Völter M. (2006) Model-Driven Software Development. John Wiley & Sons, Ltd, Chichester.

[28] Chowdhury A. & Huda M. (2011). Comparison between Adaptive Software Development and Feature Driven Development. Proceedings of 2011 International Conference on Computer Science and Network Technology, Harbin, pp. 363-367.

[29] López-Martínez J., Juárez-Ramírez R., Huertas C., Jiménez S. & Guerra-García C. (2016) Problems in the Adoption of Agile-Scrum Methodologies: A Systematic Literature Review. 2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT), Puebla, pp. 141-148.

[30] Sivonen S. (2008) Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins. VTT Publications, Espoo.

[31] Brown A. (2004) An introduction to Model Driven Architecture Part 1: MDA and today's systems. IBM, 15p.

[32] Kühne T. (2006). Matters of (meta-) modeling. Software & Systems Modeling. Springer-Verlag, Volume 5, Issue 4, pp. 369-385.

[33] Da Silva A. (2015) Model-driven engineering: A survey supported by the unified conceptual model. In Computer Languages, Systems & Structures, Volume 43, pp. 139-155.

[34] Nordstrom G., Sztipanovits J., Karsai G. & Ledeczi A. (1999) Metamodeling-rapid design and evolution of domain-specific modeling environments. Engineering of Computer-Based Systems. Proceedings. ECBS '99. IEEE Conference and Workshop on, Nashville, TN, pp. 68-74.

[35] Allilaire F., Bzivin J., Brunelire H. & Jouault F. (2006) Global Model Management in Eclipse GMT/AM3. Eclipse Technology eXchange Workshop (eTX) – a ECOOP 2006 Satellite Event, Nantes.

[36] Gjataj R. (2007) Metamodel-based Editor for Service Oriented Architecture (MED4SOA). Master thesis. University of Oslo, Department of Informatics, Oslo.

[37] Rech J. & Bunse C. (2009) Model-Driven Software Development: Integrating Quality Assurance. IGI Global, 526 p.

[38] Braga M. (read 27.12.2017). A diagram is not a model: The huge difference between them. IBM blog. URL: http://www.ibm.com/developerworks/community/blogs/invisiblethread/entry/a_diagram_is_not_a_model_the_huge_difference_between_them?=en

[39] Lundkvist T. (2011) Applications of Graph Transformation in Tools for Domain-Specific Modeling Languages. Master's Thesis. Åbo Akademi University, Department of Information technologies, Turku.

[40] Edwards G., Brun Y. & Medvidovic N. Automated Analysis and code generation for Domain-Specific Models. 2012 Joint Working IEEE/IFIP

Conference on Software Architecture and European Conference on Software Architecture, Helsinki, pp. 161-170.

[41] Pohjonen R. & Tolvanen J-P. Product Derivation through Domain-Specific Modeling: Collected Experiences. MetaCase, Jyväskylä.

[42] Kelly S. & Tolvanen J-P. (2008) Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Hoboken, N.J, p. 444.

[43] Prasanna A. (2012) A Domain Specific Modeling Language for Specifying Educational Games. Master's Thesis. Vrije Universiteit Brussel, Department of Computer Science, Brussel.

[44] Pohjonen R. & Kelly S. (2002) Domain-Specific Modeling. Dr. Dobb's Journal.

[45] MetaCase (read 24.1.2018). Benefits of MetaCASE: Nokia Mobile Phones Case Study. URL: http://www.metacase.com/papers/MetaEdit_in_Nokia.pdf

[46] Hulshout A. & Tolvanen J-P. (2007) Modeling for Full Code Generation. Embedded Computing Design. OpenSystems Publishing.

[47] Cho H. (2013) A Demonstration-Based Approach for Domain-Specific Modeling Language Creation. Doctoral Dissertation. University of Alabama, Department of Computer Science, Tuscaloosa, Alabama.

[48] Cho H., Gray J. & Syriani E. (2012) Creating Visual Domain-Specific Modeling Languages from End-User Demonstration. In proceedings of the 4th International Workshop on Modeling in Software Engineering (MISE). IEEE. Piscataway, NJ, pp. 22-28.

[49] Vatjus-Anttila J., Kreku J. & Tiensyrjä K. (2012) Domain-specific front-end for virtual system modeling. EIAC-RTESMA'12 / workshop on Graphical Modeling Language Development, Copenhagen.

[50] Tolvanen J.-P. (read 10.11.2017) Domain-Specific Modeling: How to Start Defining Your Own Language. URL: http://www.devx.com/enterprise/Article/30550

[51] van Deursen A., Klint P. & Visser J. (2000) Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Not., 35(6):26-36.

[52] Fowler M. (2010) Domain-Specific Languages. Addison-Weseley, 640 p.

[53] Selic B. (2007) A Systematic Approach to Domain-Specific Language Design using UML. 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07). Santorini Island, pp. 2-9.

[54] Karsai G., Krahn H., Pinkernell C., Rumpe B., Schindler M. & Völkel S. (2009) Design Guidelines for Domain Specific Languages. Proceedings of the

9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09) Helsinki School of Economics. TR no B-108. Orlando, Florida.

[55] Voelter M. (2013) DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. Dslbook.org, 560p.

[56] Kelly S., Lyytinen K. & Rossi M. (1996) A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. Proceedings of CAiSE'96, 8th International Conference on Advances Information System Engineering, Springer-Verlag London, UK, pp. 1-21.

[57] Pohjonen R. & Kelly S. (2007) Interactive Television Applications using MetaEdit+. Model-Driven Development Tool Implementers Forum (MDD-TIF07), TOOLS, Zurich.

[58] Brinkkemper S., Lyytinen K. & Welke R.J. (1996) Method Engineering: Principles of method construction and tool support. Springer, 324 p.

[59] MetaEdit+ (read 24.7.2017) The graphical metamodeling example. URL: www.metacase.com/support/45/manuals/graphical%20metamodeling.pdf

[60] MetaEdit+ Workbench User's Guide Version 4.5 (read 17.11.2017) URL: https://www.metacase.com/support/45/manuals/mwb/Mw.html

[61] Constatopoulos P., Mylopoulos P. & Vassiliou Y. (1996) Advanced Information Systems Engineering. 8th International Conference CAiSE'96, Herakleion, Crete. Springer-Verlag Berlin Heidelberg, 588 p.

[62] Farooji F. (2014) Evaluation of Code Generation Tools. Degree Project in Software Engineering of Distributed Systems. Department of Information and Communication Systems, Stockholm.

[63] Sandhu K.K. (1992) Specification and description language (SDL). IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications, London, pp. 3/1-3/4.

[64] Bueno D., Conger C., George A., Troxel I. & Leko A. (2007) RapidIO for Radar Processing in Advanced Space Systems. ACM Trans. Embed. Comput. Syst. 7, 1, Article 1, 38 p.

[65] RapidIO.org (read 5.1.2018). RapidIO™ Interconnect Specification part1: Input/Output Logical Specification. URL: http://www.rapidio.org/files/IO_logical.pdf

[66] RapidIO.org (read 5.1.2018). RapidIO™ Interconnect Specification part 11: Multicast Extensions Specifications. URL: http://www.rapidio.org/files/mcspec.pdf