Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model-Driven Engineering for Building Automation Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Daniel Schachinger
Matrikelnummer 0825086

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Markus Jung

Wien, 11.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model-Driven Engineering for Building Automation Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Daniel Schachinger
Registration Number 0825086

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Markus Jung

Vienna, 11.04.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Daniel Schachinger
Bauernstraße 23, 4645 Grünau

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, Wolfgang Kastner, for his steady support throughout this thesis. He did not only give me the opportunity to write my master's thesis in the Automation Systems Group, but also supported me with respect to all research- and writing-related topics. Likewise, I would like to thank Markus Jung for his continuous help and the useful hints and advice. Both were always available for answering questions and discussing relevant issues.

Moreover, my thanks go to my friends who were and are always there for me. I do not want to specifically pick out anyone as they all are amazing and supportive. Of course, my fellow students must not be left unmentioned as I have spent many great hours with them at university.

Last but not least, I would like to thank my parents, Theresia and Karl, as well as my brother Benjamin. I owe my parents my deepest gratitude for supporting me mentally but also financially throughout my life. Similarly, I can always rely on the support of my brother.

Abstract

Integrating building automation systems (BASs) into the Internet is becoming increasingly important due to the upcoming Internet of Things (IoT) paradigm. Nonetheless, a lot of different standards and technologies in the building automation (BA) sector have to be kept in mind. An automated process needs to be defined in order to enable an efficient and common way of integrating BA networks into the IoT. Gateway technologies, like Open Building Information Exchange (oBIX), offering Web services (WSs) can be used as interfaces between BASs and remote building management systems (BMSs). Model-Driven Engineering (MDE) provides a model-centric solution to establish an automated integration process by defining appropriate modeling languages and transformations.

In this underlying thesis, OMG's MDE initiative, Model-Driven Architecture (MDA), is utilized to introduce two modeling languages as metamodels. According to a four-layer architecture, these metamodels conform to the Meta Object Facility (MOF), their common meta-metamodel. A metamodel provides concepts to create models that represent snapshots of systems. The BAS metamodel is used to define platform independent models (PIMs). In this case, the BA network is mapped to abstract, technology independent models. On the other hand, the oBIX metamodel defines platform specific models (PSMs). The taken approach uses oBIX as target technology for the integration of BASs.

The transformation process, which comprises three phases, is specified based on the developed modeling languages. First, the BAS is mapped to a PIM. This step is either done manually or automatically by means of available engineering data. The actual MDA workflow starts with the existence of a PIM. Afterwards, the PIM is converted to a PSM via a model-to-model (M2M) transformation, and finally, a model-to-text (M2T) transformation generates executable source code for the target platform on the basis of the PSM.

Additionally, a proof of concept implementation, which is based on the Eclipse IDE and the oBIX integration middleware IoTSyS, is presented. The Eclipse Modeling Framework (EMF) and other MDA extensions for Eclipse (OCL Tools, Xpand, QVT Operational) are used for the realization. The different parts of the model-driven approach (models, metamodels, transformations) are separated into various Eclipse projects to form a modular structure, and the implemented concepts are evaluated by a case study. An experimental KNX network is integrated into an oBIX gateway implementation to show the functionality of the proof of concept implementation. Modeling of the BAS with the help of configuration data of the Engineering Tool Software 4 (ETS4) is discussed, as well. The functional capability of the developed, model-driven approach is pointed out in the evaluation.

Kurzfassung

Die Integration von Building Automation Systems (BASs) in das Internet wird durch das aufstrebende Konzept des Internet of Things (IoT) immer wichtiger. Es gibt jedoch eine Vielzahl von Standards und Technologien im Building Automation (BA) Sektor, die berücksichtigt werden müssen. Ein automatisierter Prozess muss definiert werden, um eine effiziente Möglichkeit zur Integration von BA-Netzwerken in das IoT zu schaffen. Gateway-Technologien wie Open Building Information Exchange (oBIX) stellen Web Services (WSs) als Schnittstelle zwischen BASs und Building Management Systems (BMSs) zur Verfügung. Model-Driven Engineering (MDE) bietet einen Modell-zentrischen Ansatz zur Erstellung eines automatisierten Integrationsprozesses auf Basis von Modellierungssprachen und Transformationen.

In dieser Arbeit wird Model-Driven Architecture (MDA), eine MDE-Initiative der OMG, eingesetzt, um zwei Modellierungssprachen in Form von Metamodellen zu entwickeln. Entsprechend einer Vier-Schichten-Architektur sind diese Metamodelle konform mit dem gemeinsamen Meta-Metamodell Meta Object Facility (MOF). Ein Metamodell stellt Konzepte zur Erstellung von Modellen zur Verfügung, die wiederum Ausschnitte von Systemen repräsentieren. Das BAS Metamodell wird zur Definition von Platform Independent Models (PIMs) verwendet. Hiermit wird das BA-Netzwerk auf abstrakte, technologieunabhängige Modelle abgebildet. Das oBIX Metamodell definiert hingegen Platform Specific Models (PSMs). Der zugrundeliegende Ansatz verwendet oBIX als Zieltechnologie für die Integration von BASs.

Der Transformationsprozess, bestehend aus drei Phasen, baut auf den entwickelten Modellierungssprachen auf. Zunächst wird das BAS auf ein PIM abgebildet, wobei dieser Schritt entweder manuell oder automatisch durchgeführt werden kann. Der eigentliche MDA-Workflow startet mit dem Vorliegen eines PIM. Anschließend wird das PIM durch eine Model-to-Model (M2M) Transformation in ein PSM konvertiert. Zuletzt erzeugt eine Model-to-Text (M2T) Transformation aus dem PSM den ausführbaren Quellcode für die Zielplattform.

Zusätzlich wird eine Proof of Concept Implementierung vorgestellt, die auf der Entwicklungsumgebung Eclipse und der oBIX Integrations-Middleware IoTSyS basiert. Das Eclipse Modeling Framework (EMF) und andere MDA-Erweiterungen für Eclipse (OCL Tools, Xpand, QVT Operational) werden zur Realisierung verwendet. Die verschiedenen Teile des Modellgetriebenen Ansatzes (Modelle, Metamodelle, Transformationen) werden in mehrere Eclipse-Projekte aufgeteilt, um eine modulare Struktur zu erzeugen, und die implementierten Konzepte werden in einer Fallstudie evaluiert. Um die Funktionsweise der Proof of Concept Implementierung zu veranschaulichen, wird ein experimentelles KNX-Netzwerk in ein oBIX-Gateway integriert. Zusätzlich wird das Modellieren des BAS mit den Daten der Engineering Tool Software 4 (ETS4) behandelt. Die Evaluierung verdeutlicht die Funktionsfähigkeit des Ansatzes.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 2
1.3 Aim of the thesis . . . 4
1.4 Methodology . . . 5
1.5 Structure of the thesis . . . 6

2 State of the art **7**
2.1 Model-Driven Engineering . . . 7
2.1.1 Overview . . . 7
2.1.2 Initiatives . . . 8
2.1.3 Principles . . . 9
2.1.4 Standards . . . 12
2.1.5 Tools and implementations . . . 13
2.1.6 Relation to ontology development . . . 14
2.2 Web services-based interfaces . . . 15
2.2.1 Overview . . . 15
2.2.2 Open Building Information Exchange . . . 16
2.2.3 BACnet/WS . . . 18
2.2.4 Devices Profile for Web Services . . . 18
2.2.5 OPC Unified Architecture . . . 18

3 Model architecture **21**
3.1 Modeling stack . . . 21
3.2 Meta-metamodel . . . 22
3.2.1 Composition . . . 22
3.2.2 Implementation and application . . . 24
3.3 Metamodel for building automation systems . . . 27
3.3.1 Construction of the modeling language . . . 27
3.3.2 Metamodel composition . . . 30
3.3.3 Model creation . . . 34
3.4 Metamodel for oBIX . . . 36
3.4.1 Adoption of the oBIX object model . . . 37

3.4.2 Metamodel utilization 38

4 Transformation process 43
4.1 Workflow description 43
4.2 Network modeling 45
4.2.1 Manual approach 45
4.2.2 Automatic approach 45
4.3 Model transformation 46
4.3.1 Mapping of metamodels 47
4.3.2 Mapping of models 51
4.4 Code generation 55

5 Implementation 61
5.1 Configuration 61
5.1.1 Development environment 62
5.1.2 Gateway implementation 65
5.2 Realization 67
5.2.1 Metamodels 70
5.2.2 Models 73
5.2.3 Model transformation 74
5.2.4 Code generation 79
5.3 Deployment 82
5.3.1 Workflow sequence 82
5.3.2 Code execution 84

6 Evaluation 85
6.1 Case study 85
6.1.1 Engineering of KNX 86
6.1.2 Experimental setup 88
6.1.3 Mapping 91
6.1.4 Evaluation 93
6.2 Related work 98
6.3 Open topics 100

7 Conclusion 101
7.1 Summary 101
7.2 Future work 102

List of Acronyms 105

List of Figures 109

List of Listings 111

List of Tables 113

Bibliography **115**

CHAPTER 1

Introduction

1.1 Motivation

The term Internet of Things (IoT) is gaining more and more attention and a lot of research is being done on this field. When reading this general term, a visionary might think about a world where cars, houses, phones are active participants integrated into the information network to simplify our everyday life [28]. Technologies like radio-frequency identification (RFID) were launched to pioneer the realization of this vision [40]. Recently developed devices already provide the required interfaces for their use in the future Internet. Furthermore, another important topic is the integration of already existing systems into the seminal concept of the IoT.

Building automation systems (BASs) are such systems which are mostly capsuled and are poorly accessible via standardized interfaces. Corresponding to the three levels of BASs [33], the devices and datapoints on the field level and the control tasks on the automation level usually communicate only with other devices in their network and with appropriate applications on the management level. Additionally, there exist a lot of different technologies like KNX [36], BACnet [5] or LonWorks [41] which make interoperability more difficult. Application developers and system integrators must be aware of the particular physical characteristics and the communication protocols of the assembled technologies [65]. Abstract and technology independent applications on the management level will only be possible, if a universal interface is provided.

A present-day building management system (BMS) is not solely limited to local execution, but needs to offer remote access more frequently. Therefore, the necessity and relevance of integrating BASs into the IoT increases steadily. Local building automation (BA) networks should be accessible from the outside via gateways to manage, monitor and control the underlying components. While the advantages of BA technologies, e.g. KNX, on the two lower levels of BASs (field level and automation level) should be kept, the management level should be extended by further functionality. There are a lot of use cases where standardized access would be beneficial. Possible scenarios could be remote control of buildings or energy analysis. Highly specialized management software, e.g. for the optimization of energy saving, could be linked easily with

different control networks behind a well-defined interface. Cost efficiency and saving of development time are only two aspects of such a concept.

Nowadays, the development of complex systems is more frequently done in a model-driven way. This approach separates the abstract functionality and the concrete implementation [35]. The Model-Driven Engineering (MDE) approach can be used to specify a common methodology relating to the mapping and integration of BASs into a model-oriented environment. This methodology will offer an independent construction of BASs and their automated transformation into particular technologies.

The field of interest is tightly related to computer and software engineering. First, the increasing significance of BASs, with their physical characteristics and low-level communication protocols on the field level and accordingly the BMSs at the management level, is an integral part of computer engineering. Second, the model-driven approach and the interfacing of BASs are settled in the field of software engineering. Finally, the integration of BASs into the IoT in the form of a transformation workflow can be seen as a combination of both computer and software engineering.

1.2 Problem statement

Since more than twenty years many different standards for BASs have been developed by various associations, organisations and companies. At present, the key players in the sector of home and building automation are KNX [36], BACnet [5] and LonWorks [41]. These standards differ in many aspects and each of them has several advantages and disadvantages depending on their field of application. Some technologies use twisted pair as communication medium, while others send and receive their data via an ordinary power line or utilize wireless protocols. Furthermore, these BAS standards vary in their topology structure, the addressing scheme or the software for engineering of the network. On the whole, there is often more than one possible technology for a certain area of application [65].

If two or more different BAS technologies are used, the BMS will have to deal with various communication protocols and other technology specific issues. The development of such a system might not only be very expensive but also time-consuming, and the resulting BMS can only be used with the current setting or limited variations of it. Therefore, the given technologies cannot be simply replaced by others. Another problem is their interoperability within one and the same application. Although technology-specific interfaces and gateways exist, there is still the disadvantage of dependence on vendors and technologies [21]. Hence, an interface is needed, which permits the management of BAS in a technology independent way. This interface has to provide an abstract view of the underlying network to enable technology independent software development. Consequently, more complex and powerful applications can be implemented. Due to the upcoming IoT, the interface should also provide standardized access to the BAS via the Internet. For example, the integration of BASs could be done by Web services (WSs) where again different technologies could be used. Open Building Information Exchange (oBIX), OPC Unified Architecture (OPC UA) and Web Services for Building Automation and Control Networks (BACnet/WS) are available and standardized ways to realize such an abstract interface [32]. The WS gateway communicates with the different BASs via the particular pro-

tocol, and its services can be accessed by standardized Web protocols like Hypertext Transfer Protocol (HTTP) or Constrained Application Protocol (CoAP).

In terms of implementing such an interface, the different automation technologies have to be mapped into a common model. A standardized and unique representation of BASs is needed in order to offer an independent and abstract access point for monitoring or control. Therefore, the multifaceted structures and schemes need to be analyzed and tailored to fit into this general model. Afterwards, the network representation in the common model can be transformed into one or more appropriate, technology specific mappings which provide the described interface.

Current research findings already provide some integration approaches for BASs, but a realization of an automated mapping from the network structure to basic WSs is still missing (cf. [22, 34, 48]). The available mappings are either implemented manually or they are focused on high-level services. On the one hand, a holistic workflow from the BA network to the interface technology is needed. This is necessary to gain access to the various BASs and provide their interoperability. On the other hand, such a workflow has to be verifiable and expandable to enable for further development. Hence, a suitable, model-driven approach is required to offer a universal way of integrating BASs into the IoT.

The concept of MDE might be a method of resolution for such an approach, since it combines both domain specific languages (DSLs) and transformation engines for these modeling languages [60]. Model-Driven Architecture (MDA), the MDE initiative of the Object Management Group (OMG), realizes this concept by supporting a four-layer modeling stack with one single and unique meta-metamodel on top. The underlying metamodels, which define the specific modeling languages, are expressed in terms of this meta-metamodel, and therefore transformations between their conforming models are supported [8]. Automatic validation and an adequate tool support are further advantages of this process. Thus, the addressed workflow from the initial modeling of the BA network to the code generation for the integration technology can be implemented using the MDA approach.

In Figure 1.1, a brief overview illustrates the model-driven workflow and the involved components. First, the network is generally represented by a platform independent model (PIM). Subsequently, this model is transformed into a platform specific model (PSM). Finally, program code is generated which is executed on a WS gateway. The modeling itself and the model transformations are part of the MDA approach. The WS gateway provides access to the network and interacts with remote clients by the use of various standardized exchange protocols and information encodings.

To sum up, based on the mentioned problem statement two hypotheses can be assembled, which are examined throughout this thesis.

Hypothesis 1 *By the use of a common meta-metamodel, modeling languages or rather metamodels can be derived to support transformations and interoperability among models of various technologies and standards.*

Hypothesis 2 *It is possible to build a fully automated transformation process for building automation systems relating to a seamless and transparent integration into a BAS technology independent interface.*

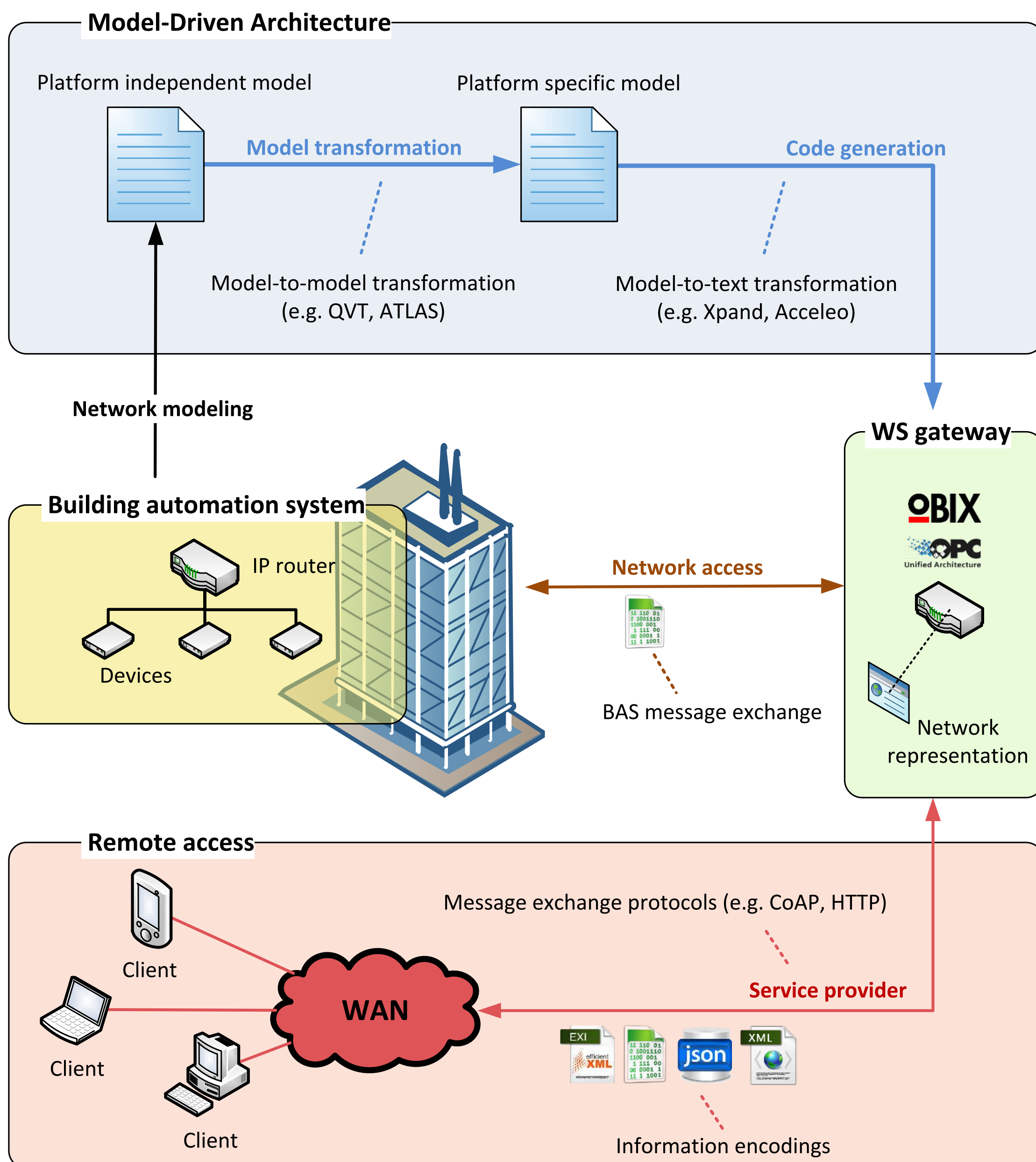

Figure 1.1: Overview of the model-driven approach

1.3 Aim of the thesis

The overall aim of the thesis is to present an approach for a seamless and automated model-based integration of traditional BAS into the IoT whereby this main goal can be divided into several subordinated objectives.

One of the outcomes of the thesis is the definition of a workflow to map a BAS in a given technology to a gateway technology, so that the system can be accessed via a standardized interface. The workflow consists of a couple of steps. First, there is the transformation from the actual network to an independent model. Second, the transformation from the abstract and independent network representation to a representation that refers to a specific WS gateway technology is shown. Finally, the last workflow step is concerned with the generation of executable code from this representation.

Additionally, a model-driven approach on top of the transformation workflow is described. The aim is to realize a universal way of integration which is independent from the implementation. The dependencies and interactions between the levels in the modeling stack and between the models within one level are stated. Moreover, the development and structure of the appropriate modeling languages or rather metamodels are pointed out.

A proof of concept implementation demonstrates the functionality of the above-named workflow and the model-driven approach. The realization of the individual components like metamodels or transformation patterns is specified. In addition, the deployment of the proof of concept implementation is demonstrated.

In the context of this master's thesis, a KNX network is used as BAS in a case study. This network is integrated into an oBIX gateway to provide access to the network via the Internet. A sample project passes through the workflow. The generated code is tested on an instance of an oBIX gateway implementation. Furthermore, a mapping of the most common KNX specific elements like datapoint types to the oBIX gateway is discussed.

Finally, the thesis contains an overview of information technology (IT)-friendly solutions and research projects to integrate BASs into the IoT as well as an overview of MDE approaches. The related work in this field of interest is examined and compared to the results of the thesis.

1.4 Methodology

Relevant literature in the field of MDE and WSs-based interfaces is studied to illustrate the concept of these areas. Based on this research, a model-based approach is designed in accordance to the MDE standard MDA of the OMG [51]. Thereby, the initial step is to outline the metamodels for the PIM and the PSM, i.e. the oBIX object model. The metamodel for the platform independent level has to be derived from schemes of existing BAS technologies. The greatest common divisor has to be found and must be expanded by other relevant model elements in order to cover the BAS specifications. Afterwards, the MDE tools are used to prepare the model-to-model (M2M) and the model-to-text (M2T) transformations. The combination of these metamodels and the corresponding transformations lay down the searched workflow for integration of BASs into the IoT.

The proof of concept implementation and its evaluation are based on a KNX installation and an already existing implementation of an oBIX gateway that is able to communicate with the connected network via Calimero [13]. The gateway's Web interface is accessible via standard Internet protocols like HTTP or CoAP. At the beginning of the integration workflow, the Engineering Tool Software 4 (ETS4) is used to export the engineering data of the network as multiple Extensible Markup Language (XML) files. These XML files conform to a KNX XML Schema

and are converted into an independent model according to the previously constructed metamodel by a mapping with Extensible Stylesheet Language Transformation (XSLT). Henceforward, the network representation is transformed into a PSM. In this thesis, oBIX is chosen as platform specific target technology. Finally, the oBIX model of the KNX network is transformed into Java source code. The implementation contains the necessary information of network structure, datapoints and devices. The source code is executed by an instance of the previously mentioned oBIX gateway where the required libraries for the generated source code can be found. The models, metamodels and transformations are created with various modeling tools of Eclipse [18].

In addition, the theoretical, model-based approach and the presented workflow implementation are reflected in terms of open topics, problems and constraints. Available related work is linked with the outcomes of this thesis to offer a coherent view with respect to the field of interest. Thus, the advantages of the elaborated approach are presented and the differences to existing research are illustrated.

1.5 Structure of the thesis

This section serves as guidline for the structure of this thesis. In Chapter 2, the state of the art concerning MDE is presented. After a short overview, the most popular initiatives of MDE are discussed, its main principles are explained in detail and standards and implementations are described. Furthermore, this chapter deals with Web services-based interfaces.

The main part of this thesis starts in Chapter 3, which shows the model architecture with the modeling stack, the description of the meta-metamodel and the specified metamodels. Chapter 4 contains the transformation process including a workflow description as well as the model transformation and the code generation.

Consecutively, Chapter 5 deals with the proof of concept implementation where a BAS network model is integrated into a WS gateway. First, the configuration of the used environment is presented, and the software development environment as well as the oBIX gateway implementation are described. In the next part, the implementation with its metamodels, models and transformation patterns is explained. Finally, the workflow sequence from the model creation to the execution of the generated source code in the oBIX gateway is specified textually and graphically. The source code of the implementation is available online [45].

In Chapter 6, the presented approach is critically reflected and compared to alternative concepts. While a case study demonstrates the functionality of the presented model-based approach by integrating a KNX network into an oBIX gateway, also open topics are discussed in this part of the thesis.

Finally, the thesis is concluded with a summary and an outlook (Chapter 7).

CHAPTER 2

State of the art

The underlying chapter comprises the theoretical background and the basic principles necessary for the development of the contemplated approach. As the thesis is concerned with the integration of BASs into an independent interface based on a model-driven approach, the first section examines Model-Driven Engineering (MDE) with an overview of underlying principles, available initiatives, defined standards and actual tools and implementations. Afterwards, some Web services-based interfaces are discussed whereby the main focus is on the Open Building Information Exchange (oBIX)[1] standard.

2.1 Model-Driven Engineering

The software development methodology called MDE relies on the utilization of models instead of traditional code-centric object-oriented technologies [64]. The basic ideas behind this concept are discussed in the following subsections.

2.1.1 Overview

In the past, the statement *"Everything is an object"* [9, p.171] has been the predominant principle in developing object-oriented technologies. Within these technologies, classes and instances of these classes (i.e. the objects) are the main elements. In addition, classes can inherit from other classes [26]. MDE, on the other hand, can be seen as a shift towards a model-centric view where the basic principle is *"Everything is a model"* [9, p.171]. In this approach, models are the main concept as they head the whole development process [30]. They represent a (real) system, and conform to a metamodel [26]. In [8] and [9], Bézivin illustrates this relation as shown in Figure 2.1. An advantage of MDE is the way of handling the increasing complexity of system development. The abstraction level is raised with the introduction of models [64].

[1] OASIS has changed the notation from *oBIX* to *OBIX* in newer versions of the standard.

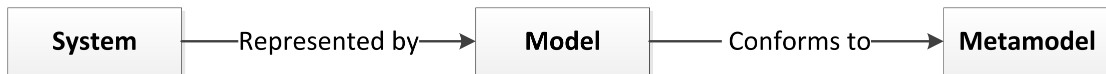

Figure 2.1: Basic notions in MDE [8, 9]

In the past, various efforts relating to hoist the abstraction level in software development have been realized. Examples are Computer-Aided Software Engineering (CASE) or object-oriented programming languages like Java or C#. However, platform complexity is still rising which is the result of e.g. the appearance of new platforms and changes in existing ones. Thus, MDE technologies are developed to address this complexity issue [60].

2.1.2 Initiatives

Two well-known initiatives, motivated by the MDE approach, exist. Both constitute applicable methods based on the MDE principles. First, the Model-Driven Architecture (MDA) approach is presented before an overview on the Eclipse Modeling Framework (EMF) is given.

2.1.2.1 Model-Driven Architecture

The Object Management Group (OMG) launched the MDA initiative by the end of 2000. One of its purposes is the shift from code orientation to model orientation in software development. The platform dependent implementations should be separated from the abstract business logic. OMG's Unified Modeling Language (UML) has driven the evolution of this approach. The concept of metamodels, which provide modeling languages for the particular models, is expanded by meta-metamodels. Thus, the independent development and evolution of non-compatible metamodels is avoided. MDA defines a four-layer architecture. On top, the meta-metamodel (1), which conforms to itself, defines the language for building metamodels (metamodeling language). One level lower, metamodels (2) are located that are used by the subjacent models (3) to create snapshots of the observed system (4). These systems are on the lowest level of the architecture. More precisely, MDA has a 3+1 architecture because the three upper levels can be defined as *modeling world* while the bottom level forms the *real world* [8].

Another basic concept of MDA is the model transformation which originates from the homonymous MDE principle (see Section 2.1.3). The first task in the MDA process is the definition of a computation independent model (CIM). This model describes the system on an abstract level [35]. Subsequently, a platform independent model (PIM) is created which represents the developed system and its functionality. PIMs are independent of any technical detail of the target platform. M2M transformations produce a platform specific model (PSM) on the basis of the PIM. Finally, a M2T transformation generates code for a target platform from the PSM [7]. In this context, a *platform* can be a closed software component or technology with a clearly defined interface. The platform provides necessary services, but its implementation does not need to be known [35]. Figure 2.2 illustrates the transformation process. The idea of having PIMs and PSMs has an important advantage. Only one PIM containing the abstract system representation is needed to generate descriptions in the form of PSMs for various successive

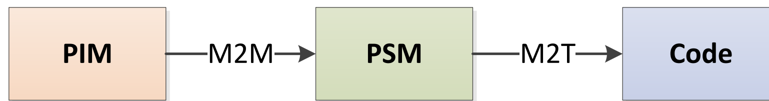

Figure 2.2: MDA process [7]

target platforms (vertical one-to-many model transformation) [44]. Specific metamodels for the modeling of PIMs and PSMs conform to one distinct meta-metamodel [8]. Hence, the system can be deployed more easily on more than one platform. If there are changes or new platform technologies emerge, the underlying abstract system model does not need to be changed [43].

It should be noted that MDA is more than a simple code generation methodology due to the formalization in modeling software architectures and platforms [56]. MDA combines various standards like Object Constraint Language (OCL), Meta Object Facility (MOF) or XML Metadata Interchange (XMI) [23].

2.1.2.2 Eclipse Modeling Framework

The EMF offers a framework for modeling and code generation which runs on the open source project *Eclipse*. It is part of the Eclipse Modeling Project and represents another initiative of the MDE concept. Similar to MDA, the basis is a metamodeling language in the form of a meta-metamodel called *Ecore*. This meta-metamodel is included in the core EMF. Individual modeling languages can be defined based on Ecore in order to enable for the creation of domain specific models. Finally, Java classes can be derived from these models. For this purpose, EMF offers generator components. In addition, modeling editors can be created based on the defined metamodels to enable the formation of models [63]. Moreover, further projects have been introduced to expand the model-driven functionality of Eclipse. These additional projects are partially based on EMF [12]. The Ecore meta-metamodel and some tools and implementations of the Eclipse Modeling Project will be presented in Section 2.1.5.

2.1.3 Principles

According to Brambilla et al. [12], the MDE intention can be expressed with the following equation: $\textit{models} + \textit{transformations} = \textit{software}$. Therefore, the models and corresponding transformations can be identified as the main elements of model-driven approaches. Section 2.1.3.1 addresses the problem of metamodeling to define modeling languages. Transformation methodology is discussed in Section 2.1.3.2 (M2M transformation) and Section 2.1.3.3 (M2T transformation).

2.1.3.1 Metamodeling

In MDE, models are not only used for documentation purposes but also as formalized components for computer-based development [9]. In this context, metamodels form a concept for the definition of modeling languages. They describe the available language constructs for the creation of models [12]. Metamodels are not necessarily specified in a standardized way, but can

also be written in a natural language (e.g. English). Nonetheless, computer-aided processing of models (e.g. validity checks and transformations) assumes a formalized definition of metamodels [26].

Regarding the syntax of a modeling language, it can be distinguished between the *abstract* syntax and the *concrete* syntax. An abstract syntax is based on the metamodel of the language and specifies the underlying elements and constructs. On the other hand, the concrete syntax defines the appearance and the notation of the models. This is determined in less formal descriptions and illustrations [56].

The outcome of metamodeling are modeling languages which allow the definition of concrete models. These representations of a system of interest have to comply with the rules specified in the metamodel. In general, two types of modeling languages exist. While domain specific languages (DSLs) are used to create models for a certain domain, general-purpose modeling languages (GPLs) are not limited to a specific domain, but can be used for any purpose. The most popular example for a GPL is the UML [12].

The layered modeling architecture has already been mentioned in Section 2.1.2.1. Typically, the systems are mapped to models which conform to a modeling language or rather to a metamodel. On top, a model for describing metamodels is defined which is called meta-metamodel. Although further layers are possible, these four layers are usually sufficient. As the meta-metamodel is defined by itself, the entire architecture is closed. By means of the metamodeling principle, the DSLs of an MDE approach can be developed to provide a basis for the creation of models and their subsequent transformations [12].

2.1.3.2 Model-to-model transformation

During an MDE development process, model transformations are used to generate new models or executable program code [30]. Brambilla et al. define some operations that are implemented as model transformations [12]. Thus, models can be merged, aligned, refactored, refined or translated. The transformations can be classified into *model-to-model (M2M) transformations* and *model-to-text (M2T) transformations* [15]. Although this section focuses on M2M transformations, the following paragraphs state the common characteristics of both types.

In general, a transformation converts input model(s) into output model(s) by executing specified transformation rules. Input models are called *source models* while output models are called *target models* [26]. A transformation rule is made up of a left hand side (LHS) and a right hand side (RHS). While the LHS represents the source model, the RHS corresponds to the target model. Each side of a rule is composed of variables (e.g. elements from the source model), patterns (e.g. model fragments) and logic (e.g. OCL queries) [15]. Admittedly, transformations are defined based on metamodels of the input and output model. However, they apply models conforming to these metamodels. The transformation itself conforms to its own metamodel which defines the transformation language. Therefore, it is a kind of model, too [12].

There are some possible classifications of model transformations. First, they can differ in their directionality. While *bidirectional* transformations can be used to run the transformation in both directions, *unidirectional* transformations execute the transformation definition in one direction only [15]. Another characteristic is the number of input and output models. There are *1-to-1* transformations with one source model and one target model. A transformation of

one source model into multiple target models is called a *1-to-N* transformation. The other way around, they are called *N-to-1* transformations. An *M-to-N* transformation is the most universal case where multiple input models are converted to multiple output models [26]. If a more abstract model is transformed into a more specific model, the process is called *vertical* transformation. On the contrary, transformations performed between models of the same abstraction level are known as *horizontal* [61]. Finally, the transformation languages can either be *declarative* or *imperative*. Declarative languages define relationships between source and target elements. While the execution order is not set in declarative transformation definitions, imperative languages specify an explicit execution plan [26].

In M2M transformations, both the input and the output are models and not any kind of text or code. Such transformations are used to generate intermediate models during the evolution of a PIM to the final program code. Therefore, the abstraction gap between PIM and code is shrinking, which enables for the easier optimization and maintenance of the transformations. There exist some approaches for M2M transformations. The *direct-manipulation* approach provides an application programming interface (API) for manipulating models (e.g. Jamda). The *relational* approaches are based on mathematical relations. Constraints are used to specify the relations between source element types and target element types. An example for such transformations is Query/View/Transformation (QVT). *Graph-transformation-based* approaches use typed, labeled and attributed graphs to define the transformation (e.g. VIATRA). OptimalJ is an example for *structure-driven* approaches which consist of two phases. First, the hierarchical structure of the output model is created, and afterwards the attributes are set. Finally, also combinations of these four approaches exist (*hybrid* approaches). An example for this category is ATLAS Transformation Language (ATL) [15].

2.1.3.3 Model-to-text transformation

The aim of M2T transformations is the generation of documentation and other text documents as well as the generation of executable source code. This corresponds with the overall goal of MDE to establish a running system on the basis of a platform independent application model. Whereas compilers generate machine code out of source code, the code generation in MDE is the transformation of a model into source code [12].

For M2T transformations three questions have to be answered. First, it has to be determined how much code can be generated (full or partial generation). Second, the kind of source code must be specified. APIs or high-level programming languages should be extensively used. Third, a procedure for generating code is required, i.e. the transformation language ranging from DSLs to GPLs needs to be defined [12].

Two approaches for M2T transformations exist. The *visitor-based* approach runs through the model representation by means of a so called visitor mechanism. In the meantime, the final code is written to a text stream. The second approach called *template-based*, which is used in this thesis, combines target text and meta code. While target text is written directly into the output file, the meta code enables access to the data of the source model [15].

In the next sections, concrete standards and implementations of these principles and previously mentioned initiatives are discussed.

2.1.4 Standards

The OMG combines a set of standards within the scope of MDA. The following subsections present standards for metamodeling, M2M transformations and M2T transformations.

2.1.4.1 Meta Object Facility

According to OMG's MOF 2.0 Core Specification, this standard enables *"the development and interoperability of model and metadata driven systems"* [52, p.5]. In MDA, MOF is one of the central technologies as it defines the concepts of metamodels, PIMs and their mapping to platforms. The standard consists of the two meta-metamodels *Essential MOF (EMOF)* and *Complete MOF (CMOF)*. While EMOF is used to specify simple metamodels by means of simple concepts, CMOF is more complicated but also more expressive [52]. The composition of MOF respectively EMOF is discussed in Section 3.2.

MOF enables the definition of modeling languages. Thus, the meta-metamodel represents a metamodeling language [12], while the meta-metamodel itself is defined by using its own language concepts [26]. In the four-layer architecture of MDA, the metamodels on the second highest layer are defined by the MOF language. Both semantics and structure are determined in the MOF standard. All in all, an advantage of this common framework is the interchangeability between models or metamodels conforming to MOF. Furthermore, the systematic integration of metamodels and models is simplified [57].

2.1.4.2 MOF Query/View/Transformation

Another standard of the MDA initiative is Query/View/Transformation (QVT) which is published by OMG, as well. The architecture of this model transformation standard consists of three DSLs. First, the *Relations* language defines relationships between models. QVT Relations is based on the *Core* language which is as powerful but smaller. While both Relations and Core are declarative languages, *Operational Mappings* is imperative [54]. Transformations written in this language are unidirectional, and their syntax is comparable to other imperative languages [6]. Hence, QVT constitutes a hybrid language which is limited to M2M transformations [26].

Besides these languages, QVT offers an additional, useful feature: *Black Box Implementations*. They enable the possibility to link the transformation with implementations and libraries written in other languages. In the transformation definition, the signatures of the operations are sufficient [54]. Therefore, the implementation can be seen as a black box. The expressiveness of other programming languages can be utilized to implement e.g. complex algorithms [6].

2.1.4.3 MOF Model to Text Transformation Language

M2T transformations are standardized by the OMG in the MOF Model to Text Transformation Language (Mof2Text). In general, the standard describes how a model can be transformed into text. Examples for generated text representations are deployment specifications, reports or documentations as well as source code. Mof2Text follows the template-based approach of M2T transformations [53].

The construction of templates is introduced in the specification. Besides target text, the templates contain meta code which is a kind of placeholder for model information. Values in the source model can be accessed via expressions and queries. Furthermore, transformations can be organized as modules. A special part of a template is the *file block* which defines the final location of the generated text. Here, the most important parameter is a Uniform Resource Identifier (URI) representing the name of the output file. Further information regarding this standard can be found in the corresponding specification [53].

2.1.5 Tools and implementations

In this subsection, some available tools and implementations of the previously mentioned standards are listed. The focus lies on technologies that are used throughout this thesis. Descriptions are kept concisely to give a rough overview. More details can be found in the underlying literature and in the following chapters.

Ecore is the meta-metamodel of the EMF, and therefore provides a metamodeling language for specifying modeling languages. This language is a main difference between EMF and MDA where MOF is defined as meta-metamodel on the topmost level of the modeling architecture [26]. Ecore arises from MOF and is similar to the slim structure of EMOF. A kernel of four classes is supported by additional language concepts [63]. The Ecore kernel is examined in Section 3.2.2.

QVT Operational (QVTO) is part of the Eclipse Modeling Project. Instead of implementing the full QVT standard, QVTO enables for the definition of transformations in the Operational Mappings language [17]. The transformations are allowed to contain concepts like loops and conditions as QVTO is an imperative language [26]. This Eclipse component is used for the M2M transformation in the model-driven approach of this thesis.

ATLAS Transformation Language (ATL) is another technology for M2M transformations. The transformation definition is a model conforming to the MOF-based ATL metamodel. In contrast to QVTO, this language is both declarative and imperative. As a hybrid approach, it is useful in cases where neither a pure declarative nor an imperative solution is constructive. In addition, ATL transformations generate write-only target models from read-only source models. Due to their unidirectionality, two transformations will be needed for a bidirectional mapping [31].

Xpand is one of the components of openArchitectureWare (oAW), which are now part of the Eclipse Modeling Project [18, 55]. The template-based technology is used to transform models into text (e.g. source code, documentation). For this purpose, the Eclipse component provides an editor for the creation of templates. In this thesis, Xpand has been selected as M2T transformation language. Samples of Xpand transformations can be found in Section 4.4 as well as online [45].

Acceleo is an alternative implementation of the Mof2Text standard. Its aim is to generate source code for various platforms based on models. Besides Xpand, it is also part of the M2T

category within the Eclipse Modeling Project [18]. As this technology is not used in this present thesis, further details are omitted.

2.1.6 Relation to ontology development

An ontology can be described as a methodology to represent and organize knowledge. In addition, ontologies provide the possibility to generate new information by *reasoning* on the available data. In the context of the Semantic Web, ontologies play a major role. Examples for ontology description languages are the Resource Description Framework Schema (RDF Schema) and the Web Ontology Language (OWL). There are some advantages when using ontologies for the representation of heterogeneous BASs. First, the BA networks can be configured in a central repository. Second, the machine-readable representation of the ontology can be used as access point by associated systems. Third, the reasoning enables the automatic generation of configuration data for gateways, which integrate the various BASs. New BA technologies can be incorporated into a network of existing BASs by implementing just one mapping for the ontology [58].

Besides MDA, ontology development is also a modeling approach. Both have some features and characteristics in common. Thus, it is possible to combine these approaches. Gašević et al. present a mapping from OWL to MDA which is depicted in Figure 2.3 [26]. The possibility to model ontologies within the MDA concept is shown. On the one hand, there is the RDF Schema modeling space as part of the Semantic Web technical space. On the other hand, there is the MOF modeling space in the MDA technical space. MOF represents the topmost layer in the MDA architecture. Beneath this meta-metamodel, a metamodel for ontologies called Ontology Definition Metamodel (ODM) has to be built. Based on OWL, it contains the general ontology concepts. In addition, an ontology UML profile is created to enable UML notation for the definition of ontologies. Mappings between the UML profile and the ODM are introduced. In the Semantic Web space, the top layer of the architecture is formed by the RDF Schema. The ontology definition language OWL is subjacent on the same layer as the MDA metamodel ODM. Horizontal mappings in both directions between OWL and ODM must be implemented. These introduce a bridge between the MOF and the RDF Schema metamodeling concepts.

Admittedly, the ontology concept seems quite similar to the MDA approach. BASs can be integrated seamlessly and the reasoning mechanism can be used effectively [58]. However, the approach in this thesis is based on the model-driven approach or rather MDA because of some advantages [26]:

- New ontology languages can be integrated easily by implementing a pair of transformations between the Ontology Definition Metamodel (ODM) and the particular technology. Otherwise, two transformations per existing ontology language have to be created. If the ontology concept is included in an MDA approach, the development is more flexible. Additional technologies can be integrated in both the ontology and the MDA space more easily. Modeling and representing of data is more abstract than direct modeling in an ontology language.

- Ontologies can be validated regarding the ODM in the MDA approach. This is important when transforming models between different modeling or ontology languages.

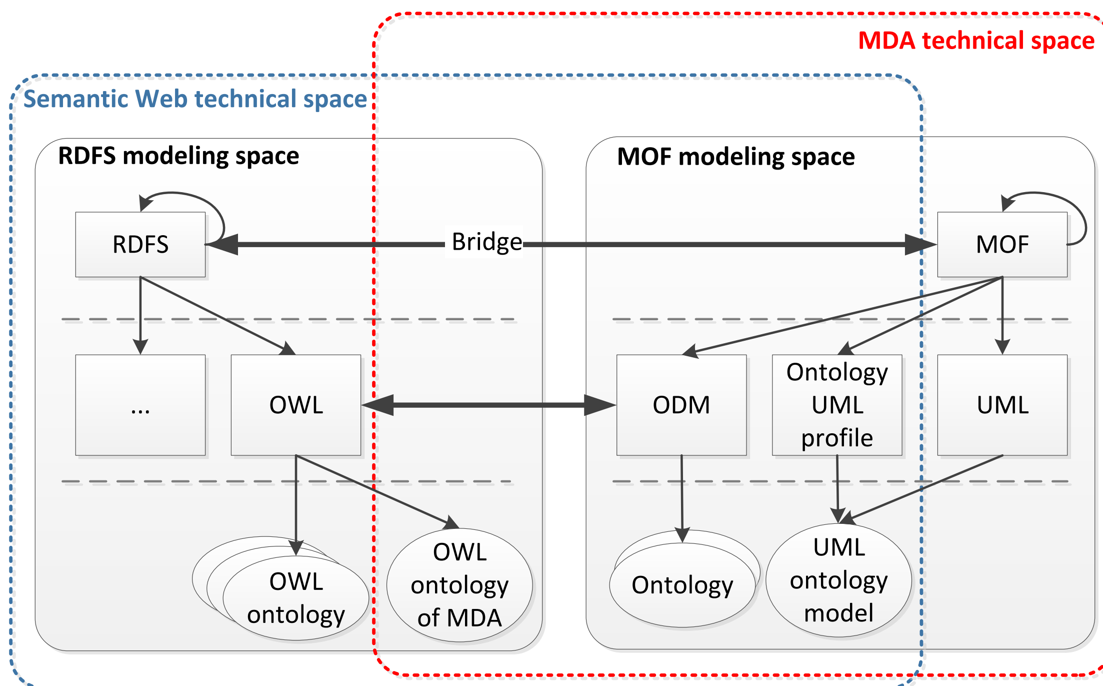

Figure 2.3: Bridging RDF Schema and MOF [26]

- ODM can be used as central metamodel for the representation of ontologies. In combination with the BAS metamodel introduced in the following chapter, this forms a common base for further processing and development.

2.2 Web services-based interfaces

As the idea of an Internet of Things (IoT) is on the edge of becoming reality, technologies for integrating heterogeneous BASs have to be found. Thus, Web services (WSs)-based interfaces could be a possible solution. After a short overview about these interfaces and the IoT paradigm, this section explains some concrete technologies in this field of application.

2.2.1 Overview

The concept of IoT claims the establishment of interconnections and interactions of *things* like sensors, actuators or mobile phones. The underlying aim is to combine these objects in order to implement enhanced functionality. Their cooperation can be used to realize common objectives. However, the accompanying security risks should not be disregarded when integrating things of everyday life in a pervasive network [4].

The IoT is the intersection of three visions. First, the *Things oriented* view targets (mostly) low-level communication devices. Second, *Internet oriented* visions include concepts for a network oriented view of the IoT. And last but not least, the *Semantic oriented* perspective covers

the issues of a unique addressing scheme for the integrated objects as well as the reasonable representation of information. A lot of possible scenarios and application domains for the IoT exist, e.g. transportation and logistics (assisted driving, augmented maps), healthcare (data collection, identification) or smart environments (comfortable homes) [4].

Middleware approaches are needed for the integration of legacy systems and the development of high-level services [4]. Here, protocols on the basis of WSs are on the rise. The choice is between WSs according to the Representational State Transfer (REST) paradigm (see Section 2.2.2.3) or Simple Object Access Protocol (SOAP) services [32]. Such WS technologies for the integration of BASs are Open Building Information Exchange (oBIX), BACnet/WS, Devices Profile for Web Services (DPWS) and OPC Unified Architecture (OPC UA), which are examined in the following sections.

2.2.2 Open Building Information Exchange

The oBIX standard has been published by the Organization for the Advancement of Structured Information Standards (OASIS). In order to avoid the use of low-level protocols while integrating embedded systems, this standard provides an IT-friendly interface using standard technologies. In the following subsections, key elements of the oBIX architecture are presented, while additional information can be found in the oBIX specification [49].

2.2.2.1 Object model

In the oBIX technology, the concise object model defines the structure of objects which can be instantiated. According to [49], Figure 2.4 shows the class diagram of this object model. The base class is `Obj` containing all common properties (e.g. `name`, `writable`, `href`). Derived objects for integers, strings, floating-point numbers or even lists and operations exist below this root object. Any complex, compound object can be created with this small set of basic objects. The attribute `obj` in `Obj` is an association which enables the creation of object hierarchies. Thus, each object can contain other objects. For example, an instance of the class `Int` can contain a `String` object and a `Real` object. Details about the various derived types and their attributes can be found in the oBIX specification [49].

Objects are identified via their `name` within a composite object. On the other hand, an object is referenced via its unique URI when making a request over the network. The URI is set in the `href` attribute of `Obj` [48]. Data types of the attributes are based on XML types [66].

2.2.2.2 Contracts

A key concept in oBIX is the definition of *contracts* which introduce inheritance. Contracts are used as templates to specify oBIX *types* and their semantic interpretation [32]. Since every oBIX element is an object, also contracts are (composite) objects. If an object is inherited from a contract, the subobjects are inherited as well. Similar to multiple inheritance in programming languages, references to multiple contracts are possible. Hence, contracts bear analogy to Java interfaces [48].

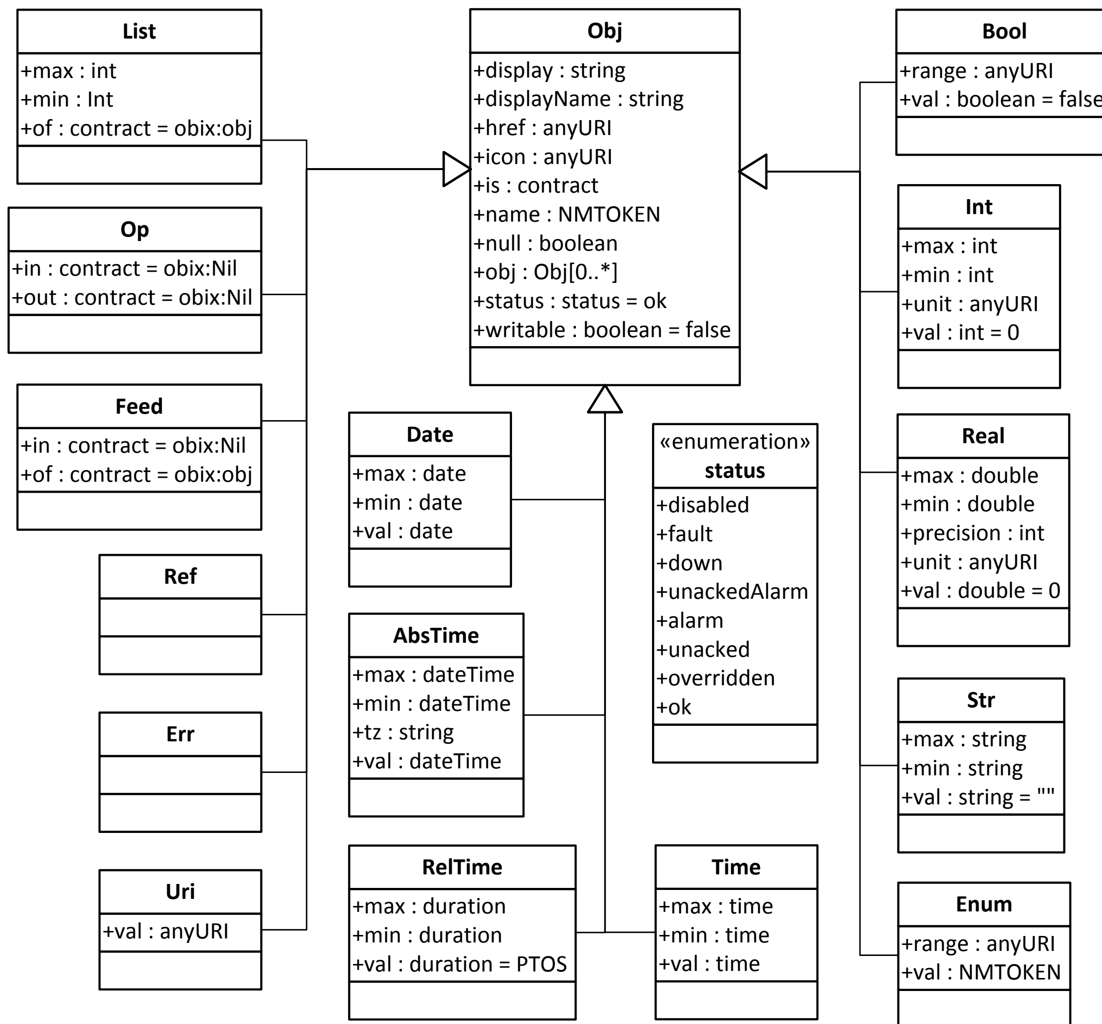

Figure 2.4: oBIX object model [49]

Contracts offer a machine readable format which can be processed automatically by clients. In addition, the flexibility and the simplicity are advantages of this concept. It can be distinguished between *explicit* and *implicit* contracts. While the explicit contract specifies the object structure, the implicit contract is focused on the semantics. In general, objects are derived from contracts by setting the value of the `is` attribute to the URI of the contract. The type of a `List` object is set in the `of` attribute, and the input and output parameters of an `Op` object are specified in its `in` and `out` attribute. Moreover, lists of contracts can be assigned where the contracts are separated by a blank. The default values of the subobjects of the contract are inherited unless the instantiated object defines a child object with the same name. Then, the attributes are overridden by the new values. Furthermore, the oBIX specification introduces a core contract library (e.g. unit, range) [49].

2.2.2.3 REST paradigm

REST is an architectural style for WSs. The key characteristic is its resource orientation. A small set of operations is available to access resources. This approach is comparable to the concept of the World Wide Web (WWW) [48]. A resource, which is an oBIX object, is accessed via its specified URI. Mainly, three operations (verbs) are used to work with all oBIX resources. While a `read` request is applicable to any object, *write* is only for writable objects. The former is the counterpart to the HTTP `GET` whereas the latter is mapped to HTTP `PUT`. Operations can be executed by an `invoke` request similar to HTTP `POST`. In addition, there exists a fourth REST verb called *delete* [49].

2.2.3 BACnet/WS

BACnet/WS is an amendment to the BACnet standard published by the American Society of Heating, Refrigerating and Air Conditioning Engineers (ASHRAE) [2]. The aim is the specification of a WS interface and an appropriate model to integrate BASs in enterprise BMSs. The interface is designed in accordance to the service-oriented architecture (SOA) principle. BA technology independent services for managing data use XML in connection with SOAP based on HTTP. The available services can be grouped in reading and writing services [34].

The main element of the BACnet/WS data model is the *node*. Nodes can build hierarchical structures, and contain attributes. Attributes are categorized in primitive, enumerated and array attributes. The paths for identifying nodes and attributes consist of a node part and an attribute part. Nodes are linked by references. Here, loops and self referencing is forbidden. In fact, the data model is simple but not extensible [34].

2.2.4 Devices Profile for Web Services

Another SOA protocol especially for embedded devices is the DPWS. As successor of Universal Plug and Play (UPnP), it is independent of BA technologies, as well. Various protocols are utilized by this profile. Examples are HTTP, XML or SOAP [62]. Windows Vista and Windows 7 already include DPWS [14].

The DPWS middleware provides WSs for the integrated devices and their hosted services which enable access to the device's functionality. In addition to the hosted services, DPWS specifies some core protocols. For the dynamic discovery of devices, the *WS-Discovery* is established. *WS-Addressing* is used to pack address information into the SOAP header of the messages. Services for metadata exchange provide WS metadata e.g. in the form of XML Schema definitions (*WS-MetadataExchange*). By means of the *WS-Eventing* services, devices can subscribe for receiving event notification messages from other devices. Additionally, DPWS defines *WS-Policy* and *WS-Security* to specify WS policies, and guarantee a secure message exchange [14, 62].

2.2.5 OPC Unified Architecture

The predecessor of OPC UA, OLE for Process Control (OPC), was published in the 1990s by the OPC Foundation. Admittedly, the interoperability between different technologies in BASs

was simplified. Vendors implement drivers to interact with the OPC API. The API of OPC was based on Microsoft Component Object Model (COM) or Distributed COM (DCOM). However, the dependency on this proprietary technologies and the resulting commitment with Microsoft Windows were restrictions. Hence, OPC UA has been published to replace OPC. Now, a SOA in the form of WSs is used as data transport technology to reach platform independence [22]. The loose connectivity in SOA offers an expanded applicability. Standard WS tools can be used to access an OPC UA server. Security and reliability are also part of OPC UA [27].

The available server objects are located in the so called *address space* which standardizes the representation of objects. This address space can be constructed by means of OPC UA information models [27]. In contrast to OPC, OPC UA enables the modeling of semantics besides the representation of process data. Application specific information models can be created on the same base model. Complex data can be interpreted by clients via the semantics contained in the information models. Vendor specific extensions of the base information model are also possible [22].

SOA provides a set of services for the interaction of servers and clients. The OPC UA specification groups the services in profiles which are implemented by the server [27]. The service definition is independent of any platform or protocol. Examples for available service sets are the *Discovery service set* or the *View service set*. Attributes of nodes are accessed via services from the *Attribute service set* including the `Read` and `Write` service [22].

CHAPTER 3

Model architecture

This chapter contains the structure of the MDA approach including the concept of the different modeling layers as well as the composition and utilization of the developed metamodels.

3.1 Modeling stack

An appropriate hierarchy of levels, a so called modeling stack, has to be defined when initiating an MDE implementation. In its standards, the OMG refers to a four-layer modeling infrastructure [3, 8]. Nonetheless, the number of used stack levels is not fixed and always depends on the intended use [52]. Although any number of layers can be handled, the model-driven approach in this thesis comprises of a standard, four-layer modeling architecture which is illustrated in Figure 3.1. Compared to some other representations of the OMG metamodel architecture, the numeration starts with 0 on the lowest level, and increments bottom up.

The language for defining metamodels is located in level M_3, on top of the stack. This common language, encoded in the form of a meta-metamodel, enables an interoperability of the underlying metamodels. As we already know from Section 2.1, the meta-metamodel is defined by itself. Therefore, this hierarchy is closed and no level exists above M_3. Level M_2 contains all necessary metamodels for the establishment of BASs. These metamodels are described in detail in the upcoming Sections 3.3 and 3.4. In general, metamodels are defined in terms of the overlying meta-metamodel and are introduced to determine DSLs, which describe a particular domain of interest. The models can be found on level M_1 in the modeling stack. Each model conforms to one of the defined metamodels on level M_2. Thus, they can be validated with regard to syntactical correctness and compliance with specified semantical constraints [8].

Instances of the models respectively the modeled parts of the real world are located in the lowest level, M_0. The BAS and the execution of generated program code belong to this level whereas the representation of a system, e.g. a BAS model, forms a part of level M_1 [10].

The BAS in layer M_0 is predefined as well as the meta-metamodel on layer M_3 that defines a kind of meta language. The intermediary levels have to be elaborated. All metamodels con-

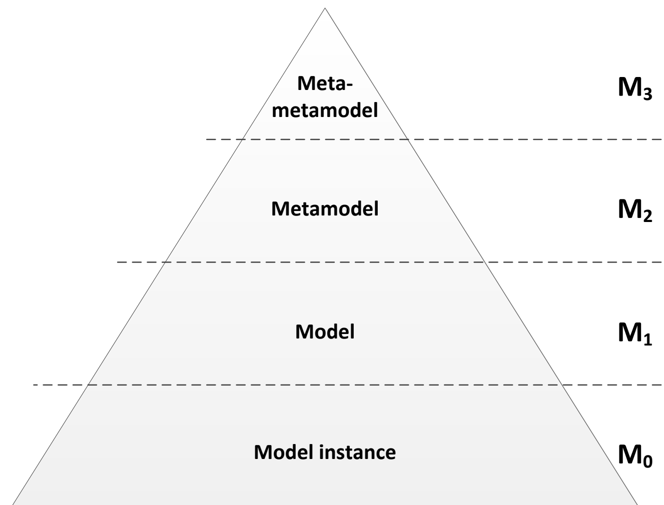

Figure 3.1: Modeling stack [8]

form to one unique meta-metamodel, and all models of one modeling language conform to the corresponding metamodel.

3.2 Meta-metamodel

As already noted, the meta-metamodel defines a meta language which is used to specify modeling languages. In OMG's MDA, this meta-metamodel on level M_3 of the predefined modeling stack is specified by the Meta Object Facility (MOF) standard [52]. This section gives an overview of the composition and the usage of this framework. Thereby, the focus is on the Essential MOF (EMOF) which is only a subset of the MOF 2.0 standard. Additionally, MOF 2.0 defines the Complete MOF (CMOF). In this thesis, the Ecore meta-metamodel of the EMF is used as implementation of MOF [63].

Ecore and EMOF are sufficient for the underlying model-driven approach as all elements of the MOF standard necessary to build the metamodels and transformations for BASs are included. Moreover, no notable implementation of CMOF exists which provides such a wide-ranging tool chain as the Eclipse Modeling Project.

3.2.1 Composition

According to [52], EMOF comprises simple concepts to build simple metamodels. First, the core classes of EMOF are illustrated in Figure 3.2 in order to start pointing out the principles of this meta-metamodel. These core classes and the subsequent figures are taken from the EMOF part of the MOF 2.0 specification [52]. Available modeling elements for the definition of the meta language are classes, properties and operations to describe the classes, and associations between the classes including inheritance and containment. The core set of classes enables for the modeling of domain specific classes (`Class`) that contain properties (`Property`) and

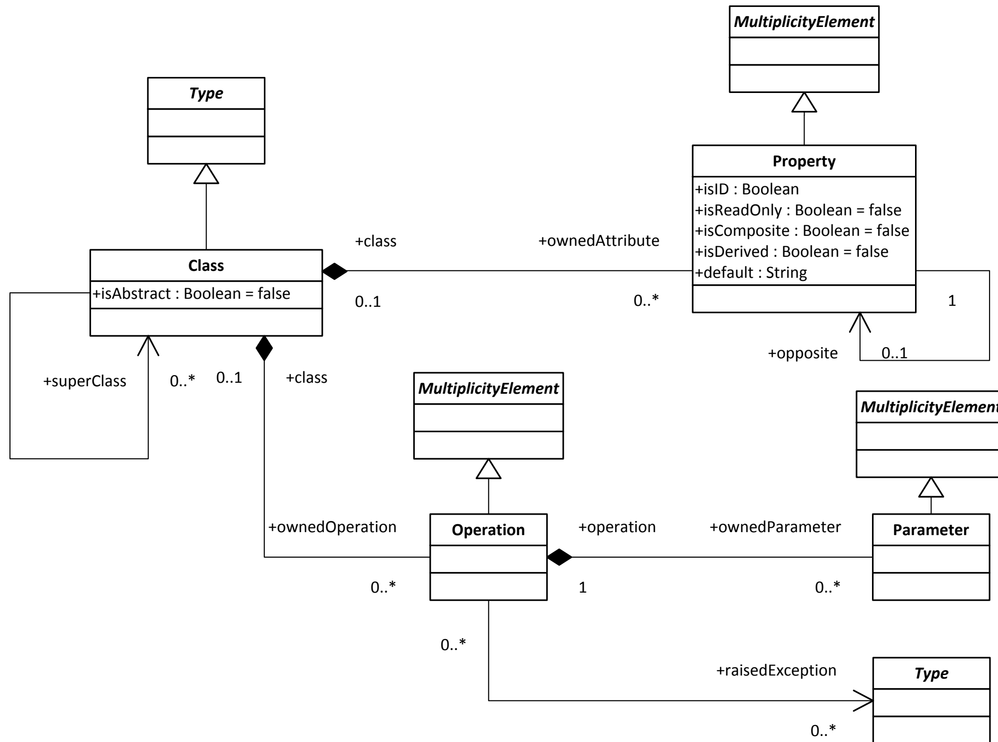

Figure 3.2: EMOF core classes [52]

operations (`Operation`). The classes can exist in an inheritance hierarchy modeled by the `superClass` association. Operations contain parameters (`Parameter`) and raise exceptions which are of the abstract type `Type`. In the subsequent figures, other EMOF structures are pointed out.

Relations between classes are modeled by means of properties. Each property has an attribute `type` derived from its superclasses `MultiplicityElement` and `TypedElement` in which the referenced type can be specified (see Figure 3.4). In contrast, CMOF defines a separate class for associations [52].

Figure 3.3 shows the composition of types and data types. EMOF defines four primitive data types (`Integer`, `Boolean`, `String`, `UnlimitedNatural`). Moreover, it supports the creation of custom enumerations (`Enumeration` and `EnumerationLiteral`) and further primitive types (`PrimitiveType`).

The topmost superclass in EMOF is the class `Element` which is depicted in Figure 3.4. Class `Object` is inherited from it and defines some fundamental operations. Named elements (`NamedElement`) are derived from `Object` and pass their properties (e.g. `name`) on to a set of subclasses (e.g. types, typed elements and enumeration literals). In addition, Figure 3.4 also introduces the multiplicity element (`MultiplicityElement`), which is the superclass of parameters, operations and properties. Therewith, the lower and upper bounds of an object's multiplicity can be modeled.

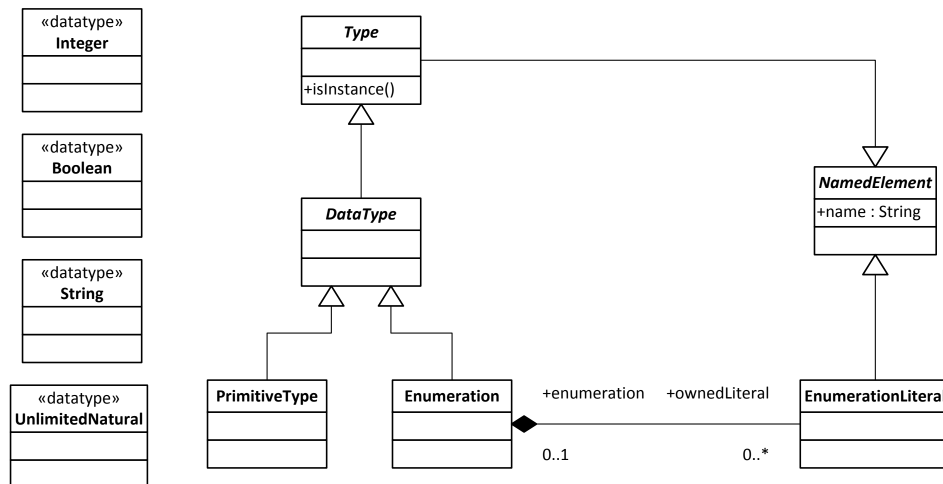

Figure 3.3: EMOF data types [52]

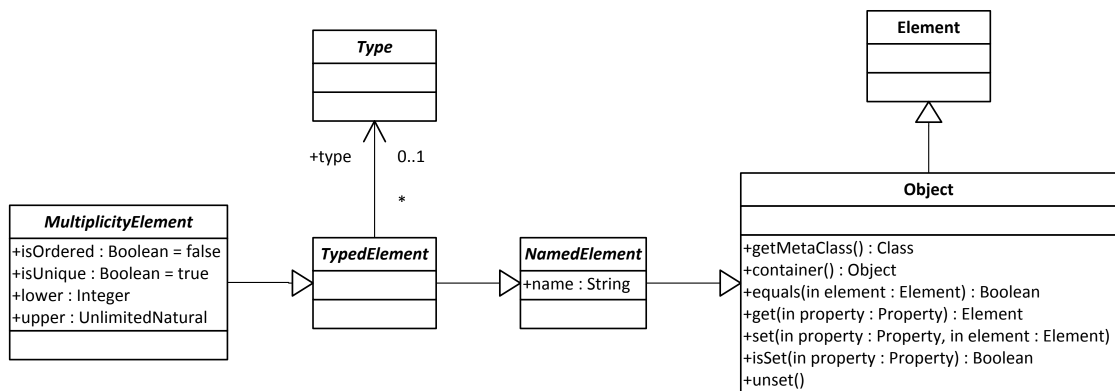

Figure 3.4: EMOF types [52]

Finally, all subclasses of the class `Type` (e.g. classes, data types, enumerations) can be capsuled in nested packages, and each package (`Package`) is a subclass of `NamedElement`. The corresponding class diagram can be found in Figure 3.5.

3.2.2 Implementation and application

The introduction of a meta-metamodel is beneficial. One of these benefits is outlined in Hypothesis 1 (see Chapter 1). If all metamodels in an MDA project conform to one unique meta-metamodel, these metamodels can be seen as instances of one and the same meta language. Therefore, elements and concepts in each metamodel are expressed in terms of the superordinated meta-metamodel. Admittedly, it is not compulsory to establish modeling languages derived from a common meta-metamodel. It is also possible to create transformations between instances (models) of almost independent languages. However, comparability of these meta-

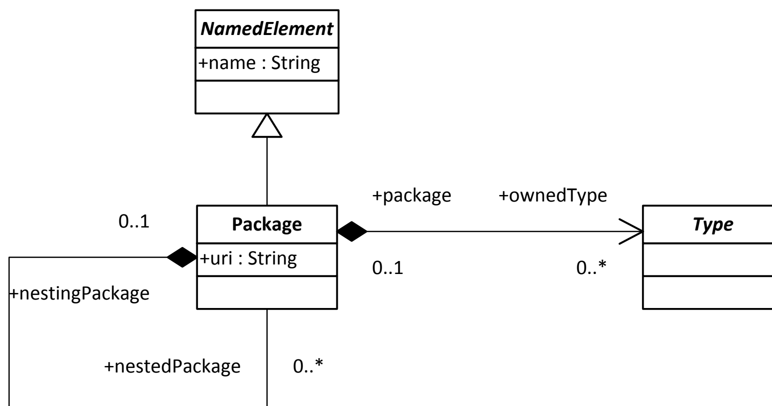

Figure 3.5: EMOF packages [52]

models and homogeneous handling are definitely advantages of implementing an appropriate modeling stack with one meta-metamodel. And these properties can be achieved by already available standards unified in OMG's MDA initiative.

The EMF for Java provides a native implementation of the EMOF meta-metamodel called Ecore. By using this framework, it is possible to create custom metamodels for the definition of DSLs. Based on these language specifications, it is possible to generate models representing snapshots of the real world. However, a mapping from MOF to EMF is needed [16]. The kernel of the Ecore meta-metamodel with its four main elements is presented in the following itemization [63]:

- `EClass` is the EMF equivalent to MOF's `Class`. Classes can contain multiple attributes and references. They support inheritance by the relation `eSuperTypes`. The value of the attribute `name` is the unique identifier of a class.
- `EAttribute` represents attributes which are components of classes. An attribute has a type and an identifying name.
- `EDataType` can be used to define types which are linked with primitive Java data types or Java classes (complex data types).
- `EReference` enables for the modeling of associations between available classes. A class can have multiple references and each reference has a referenced type which is a class again. For bidirectional associations, a second reference has to be established, since one reference navigates only in one direction. A reference can also be defined as containment (strong association).

The Ecore kernel is depicted in Figure 3.6. It should be noted that this figure visualizes only the core of EMF's meta-metamodel. Besides the four kernel elements, Ecore covers packages (`EPackage`), enumerations (`EEnum` and `EEnumLiteral`), operations with parameters (`EOperation` and `EParameter`) and some abstract classes like named elements

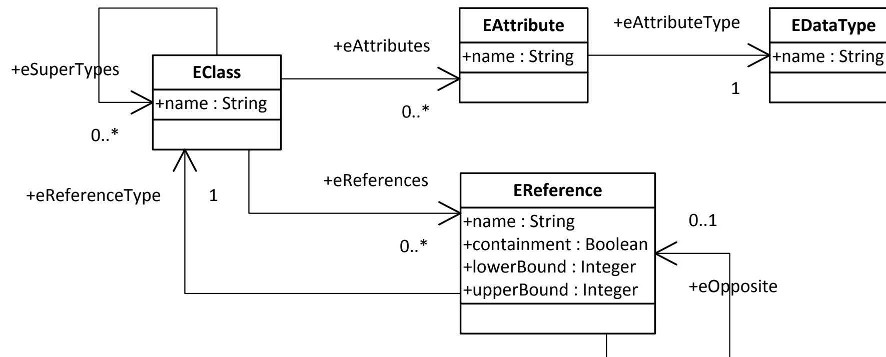

Figure 3.6: Ecore kernel [63]

Ecore data type	Java class or primitive type
EBoolean	boolean
EChar	char
EDouble	double
EFloat	float
EInt	int
EByteObject	java.lang.Byte
EBigDecimal	java.math.BigDecimal
EBigInteger	java.math.BigInteger
EDate	java.util.Date
EJavaObject	java.lang.Object
EString	java.lang.String

Table 3.1: Ecore data type mapping [63]

(`ENamedElement`) or typed elements (`ETypedElement`) in addition [63]. Only a subset of the available elements is used for modeling the metamodels in this thesis. As EMF is a Java framework, the Ecore data types have to be mapped to corresponding Java types. Table 3.1 lists the mapping of the most important data types.

To sum up, the structure of the MDA meta-metamodeling standard MOF or rather its subset EMOF, and how this standard is implemented by the EMF and its meta-metamodel Ecore has been described in detail. As already mentioned, this meta-metamodel can be used as the language description to define languages for a specific domain in the form of metamodels. Just as a UML class diagram, which represents an instance of the UML language description, metamodels use the aforesaid elements of meta-metamodels to create e.g. classes, operations, attributes or associations between classes. In the subsequent sections, the developed metamodels for the integration workflow of BASs are outlined in detail.

3.3 Metamodel for building automation systems

In the underlying, model-driven approach, the metamodel for BASs defines a language for PIMs in terms of OMG's MDA. As we know from Section 2.1.2.1, the PIM represents a snapshot of a system in an abstract, technology independent way. The following subsections document the development process of the BAS metamodel, its final structure and the possibilities to create models based on this metamodel.

3.3.1 Construction of the modeling language

The first step to build a metamodel for BASs is to establish the structure, the features and the dynamic properties of such a system. In a BA network, various sensors and actuators are linked in a specific topology. These links are realized via diverse media types, such as twisted pair, power line or radio frequency. Such a kind of network contains a set of *devices* that send and receive data over a wired or a wireless link. On the other hand, devices consist of one or multiple input/output (I/O) points, e.g. a temperature value or binary value of a light switch. These I/O points, the so called *datapoints*, are the basis for providing the behavioral aspects of devices. They are linked to other datapoints via a virtual connection based on the physical links between the devices. The semantical interpretation of exchanged messages is supported by a set of meta-data. Data types, encodings of status values, units or the scaling of numeric values are examples of semantics in BASs.

Figure 3.7 illustrates a sample BA network with a few actuators and sensors. The solid line represents the physical link, while the sum of all links, devices and routing devices is the entire topology. The dotted lines visualize the virtual links between datapoints of the devices. On top of the network's field and automation level, a BMS manages and monitors the system.

Based on these physical connections, the network can be examined by different *views*. Varying classifications of the devices and datapoints result in four distinct views. Depending on the intended use, a network can be categorized in terms of

- the *topology* structure,
- the related *domains*,
- the actual location in a *building* or building part and
- the *functionality* of the devices and their datapoints.

Besides this static structure and the various views, a BAS is characterized by its dynamic message exchange. Devices send and receive data by the use of a predefined communication protocol. As the dynamic portion of a BAS technology is not as necessary as the physical composition for modeling such a system, it is omitted in this thesis. If the network is finally modeled and a generated code is running on a gateway server, the data exchange and the communication protocol will come to the fore again. Then, the gateway must be able to communicate with the connected network, but the user (machine or human) behind the interface is only faced with the datapoints, devices and their different views. Therefore, the consideration of dynamic behavior is out of focus.

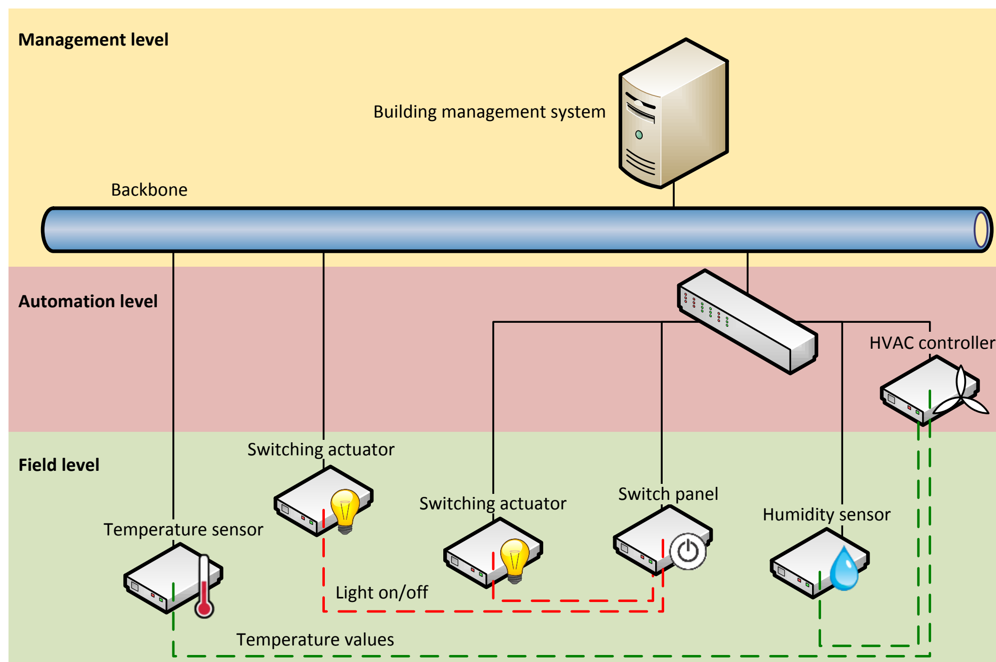

Figure 3.7: Sample automation network

Although the constructed metamodel for BASs (*BAS metamodel*) should be as general and abstract as possible to enable the mapping of lots of different BAS technologies and standards, the construction of this metamodel is inspired by the KNX technology. However, the expressiveness and universality of the defined modeling language should not be restricted. This approach combines the concept of KNX with general and abstract considerations about the fundamental structure of BA networks.

Since it is known, *what* has to be modeled, it has to be decided, *how* a metamodel for this model can be defined. The metamodel must include the network with all its relevant information in addition to the concepts for modeling metadata (e.g. types and units).

3.3.1.1 Network structure

The *network* itself is the root element of the entire model. Therefore, it is also the main class in the metamodel. The network comprises a list of *devices* and each device contains several *datapoints*. In terms of generalization, these two lists of devices and of datapoints are included separately. Thus, the metamodeling concept enables the modeling of datapoints in a system representation without providing information about the hosting devices. The link between a device and its datapoints is realized by a reference from a device to a datapoint. Although the definition of devices is universally applicable, the more generalized term *entity* is used from now on. Entity is a more abstract denomination than device.

After adapting the metamodel to integrate entities and corresponding datapoints within a network, the actual *topology* can be modeled. In KNX, the top element of the topology is a

backbone line which can contain up to 15 areas of which each one can be composed of one main line and up to 15 lines. The lines can include 255 devices besides the coupler [37]. Thus, the topology is hierarchically structured. This hierarchy of areas, lines and devices is generalized in this approach. The topology structure consists of *areas*. Each area can consist of subareas in a recursive manner. Hence, all possible, tree based and bus based structures can be modeled due to this notion. Entities can be linked with every area on every hierarchy level of the topology. In comparison to KNX, there are no limitations regarding the number of areas or linked entities in the BAS metamodel.

With respect to the used functionality in management level applications, the network and its components can be seen in different ways. One possibility is the categorization in *domains*. Again, domains can construct a hierarchy of subdomains and each domain (e.g. a lighting domain) can contain a list of references to already existing entities. Thus, it is possible to combine entities within a domain which belong together in terms of a common field of application. Another type of categorization is the separation into *buildings* and building parts. Thereby, the entities (e.g. devices) can be organized by linking them with elements of an actual building structure. Each (building) *part* has a type for characterization (e.g. building, floor, room). The construction of buildings is the same as the construction of domains. A building part can contain subparts, and these subparts can have subordinated parts.

All these views are focused on devices respectively entities, but the functionality in a BAS is based on datapoints and their message exchange for reading and writing values of sensors and actuators. Thus, the last view, which comes in mind when thinking of BA networks, is based on the *functional* behavior. Similar to the other views, there exists one main structuring element, the *group*, which can be used to form a hierarchical structure of multiple levels of groups. The difference to the previous views is that groups contain a list of datapoints and not a list of entities. An example is the group of lights and light switches in a room. It covers the particular datapoints of lighting while other groups may contain the datapoints for heating, ventilation and air conditioning (HVAC).

To sum up, these views in combination with the list of entities and the list of datapoints are sufficient to support a wide range of management activities to control a particular network.

As the elements of a network contain textual descriptions, names of vendors or other texts, a redundancy and possible inconsistency can arise. To avoid the multiple storage of the same text, the metamodel should provide classes for storing texts only once in a set of *references*. Afterwards, other elements (e.g. entities) can refer to these texts. By means of multilingual applications, the modeling of translations is also required. A translation belongs to a single element (e.g. naming of a datapoint) and is specified by a language (e.g. German, English). KNX supports multilingual texts for datapoints, devices and some metadata like units and enumeration literals. The BAS metamodel provides the modeling of translations not only for these elements, but for a lot of other elements (e.g. areas, parts, groups) as well.

3.3.1.2 Metadata modeling

Besides the definition of modeling elements for the BA network, the metamodel has to support the modeling of (data) *types*. These types are used to link raw data of the datapoints' message exchange with an appropriate semantic interpretation. A datapoint is supposed to have exactly

one type. Otherwise, it might lead to inconsistent interpretations. Two approaches for the definition of a type concept in the metamodel exist. An inheritance hierarchy can be introduced to create even more specialized types derived from general ones. However, this approach limits the possibility to expand the available range of types. Hence, it has to be thought of another solution that enables for more flexibility. Therefore, a type can be defined as a container of individual properties like an integer value, a date or a binary value. Thus, any complex data type can be created on the model level without changing the metamodel as long as the type contains only the available properties. The expressiveness of this concept is further examined in Section 3.3.2. Another advantage of this second, more flexible approach is the definition of complex parameters as input and output types of operations. Otherwise, these parameters would be limited to the metamodel types, as well.

Some of the type properties like an integer or real value need a unit. The raw value of a datapoint is useless without a unit. Otherwise, it is not known whether this value is e.g. a temperature in K or a distance in m. Therefore, *units* have to be defined in the modeling language of BASs. They contain an offset value, a scaling and a dimension (e.g. m^2).

The meta-metamodel already enables the definition of enumerations on the metamodel layer, but this definition will not be available in the models on level M_1. Thus, classes for *enumerations* have to be created in the metamodel corresponding to them in the meta-metamodel.

3.3.2 Metamodel composition

The previously mentioned considerations result in a metamodel which is presented in detail in this section. The metamodel is designed to model BASs independent from a specific BAS technology and a subsequent integration technology. The complete metamodel is illustrated in Figure 3.8. The whole set of classes can be roughly grouped in a few main parts.

The figure shows a flat inheritance hierarchy with the class `element` as superclass. The attributes `id`, `name` and `description` are needed in many classes throughout the model. Therefore, these attributes are merged in an abstract superclass. Derived classes can be identified by the attribute `id`. For multilingual support the class `element` refers to the class `translation`. Thus, a translation text for the attributes `name` and `description` can be stored for different languages.

One of the derived classes of `element` is `unit` with the additional attributes `symbol`, `offset` and `scale`. Modeled units are a form of metadata to provide semantic information for some type properties. Each unit is expressed in terms of the International System of Units (SI) to enable conversions between various units. The seven SI units are integer attributes of the class `dimension` whereby the values of these integers represent the exponents of the SI units in this particular BAS unit. A current scalar value in a given unit can be normalized by the equation $x_{normal} = x_{current} * \textit{scale} + \textit{offset}$ [49]. Additionally, a unit can contain translations since it is an ordinary element.

The diagram defines a few enumerations which can be used as attribute types in the metamodel. Additionally, it provides classes for designing custom enumerations. The former are listed on the right hand side of the figure. Translations need the enumeration `enumLanguage`, which specifies the supported languages, and `enumTranslation`, which defines the translated attribute of an element. The others are discussed later on in this section. On the other

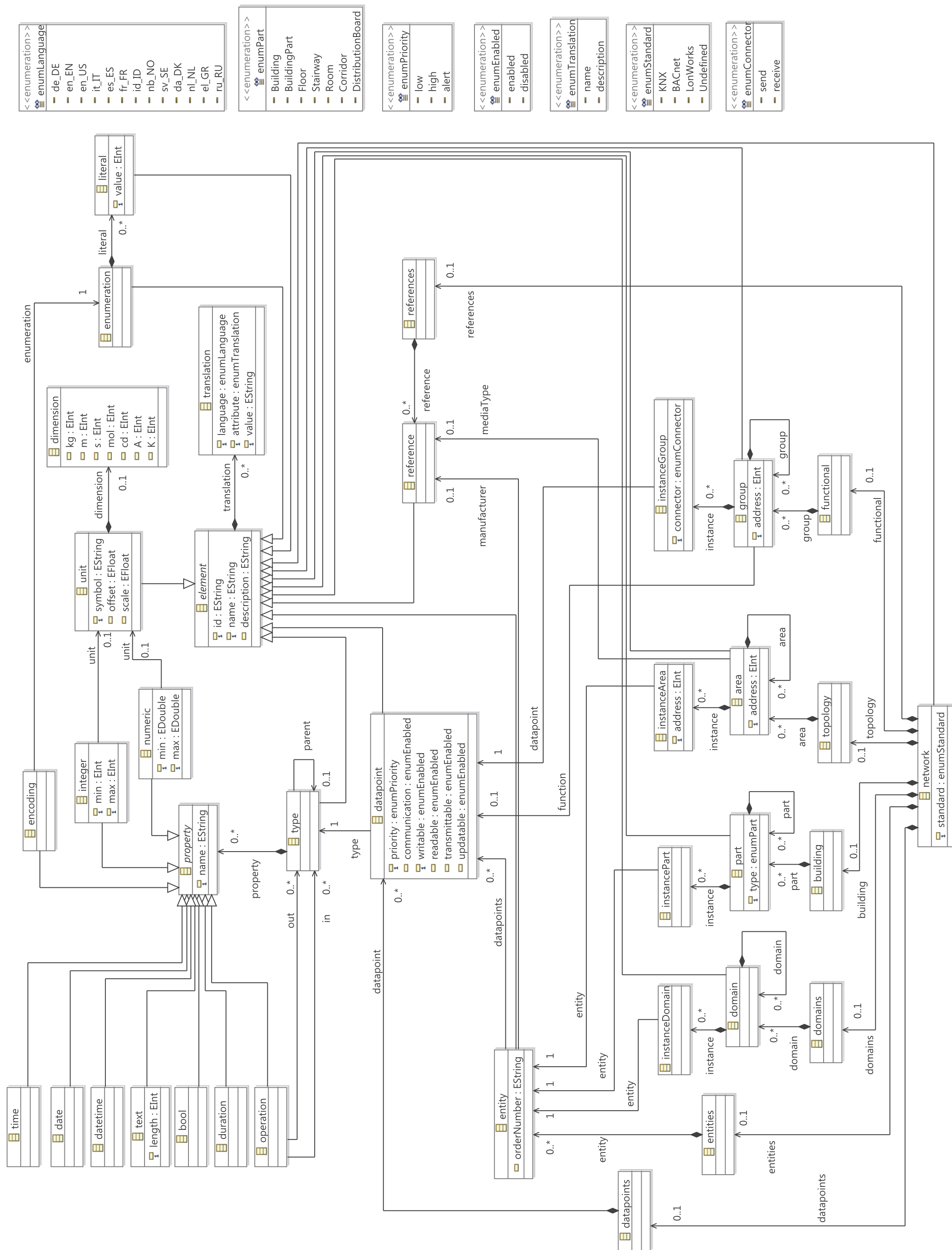

Figure 3.8: BAS metamodel

hand, custom enumerations (enumeration) contain a list of literals (literal), and each literal has an integer value which links the textual description with a machine readable value. In addition, an OCL invariant is introduced to enforce nonambiguous literal values. Listing 3.1 shows this invariant in lines 4 and 5.

```
1 class enumeration extends element
2 {
3   property literal : literal[*] { ordered composes };
4   invariant unique_literals:
5     self.literal->forAll(e1, e2 | (e1 <> e2) implies (e1.value <> e2.value));
6 }
```

Listing 3.1: OCL invariant for enumeration literals

In the upper left corner of the diagram the classes for defining data types are located. The class type has an association to itself which enables the creation of type hierarchies. The OCL invariant in Listing 3.2 prevents a type from self-referencing. As many other classes in this metamodel, a type is derived from the class element. It is a container for various properties, so that any complex data type can be designed by this language construct. Property is an abstract class with a name and has many subclasses representing various concrete property types. This set of special properties is approved by the KNX datapoint types. Any datapoint type of the KNX specification can be mapped to this concept. It would be also possible to use an enumeration attribute in the class property to define the type of the property. But this would not allow individual attributes in particular properties, e.g. the attribute length in the property class text. A special case is the property class operation which can contain references to input and output parameters (in and out). Besides a few simple property classes, there exists the class encoding which contains a reference to an enumeration. Based on this, data types can be linked with enumerations, e.g. to specify the semantics of a status byte or the meaning of a binary value (e.g. literal *on* corresponds with binary value *true*). Last but not least, the classes integer and numeric define a permitted range via the attributes min and max, and they refer to the class unit.

```
1 class type extends element
2 {
3   property parent : type[?];
4   property property : property[*] { ordered composes };
5   invariant self_parent:
6     self.parent <> self;
7 }
```

Listing 3.2: OCL invariant for type hierarchy

The key elements of the BAS metamodel are the network itself, datapoints, entities and the other views. Initially, the root element of every BAS is represented by the class network. Again, a network is inherited from element. Additionally, it provides the attribute standard

of the type enumStandard to define the BAS technology of the current network. The network refers to further BAS classes. First, the class datapoints contains a list of datapoints due to an association to the class datapoint. This is the main entry point for functionality of a BA network. Each datapoint refers to a type and is inherited from the class element. The following enumeration lists the attributes of a datapoint. The given flags are mostly derived from the KNX communication flags [25], but can be applied to other particular technologies or to a more general BA network.

- Priority states the communication priority with the available values low, high and alert which are defined in enumPriority.
- Communication is used as the main flag for activating or deactivating the communication to and from the observed datapoint. All following flags, including this one, are of the type enumEnabled, and thus they allow the values enabled and disabled.
- Writable enables for changing of datapoint values by receiving appropriate messages.
- Readable, on the other hand, allows reading of datapoint values.
- Transmittable is used to permit the datapoint to send messages on the communication medium.
- Updatable will be set (i.e. enabled), if the datapoint has to react to responses on read requests of other communication partners.

Furthermore, the network contains a list of all integrated entities accessible via the class entities which is a container of instances of the class entity. The specific information for entities is located within this class (e.g. orderNumber) or is linked by associations (e.g. manufacturer). Each entity can relate to a number of datapoints. In addition to the list of datapoints and entities, the network includes four other views. First, the view domains comprises of a hierarchy of domains and subdomains realized by the class domain and a containment association to itself. Besides this hierarchical structuring into various areas of application (e.g. HVAC, lighting), every domain on every level of this tree can contain one or more instances (instanceDomain) which refer to an entity. Second, the building view is hosted in the class building and its subordinated class part. Similarly to the domains view, hierarchical structures can be composed of parts and each part can have instances referring to an entity (instancePart). This view represents the location of the entities (e.g. devices) within a building. Building parts are characterized by the attribute type which allows values from the enumeration enumPart (e.g. building, floor, room). Third, the network topology is illustrated by the view topology. As it can be seen in the metamodel figure, a tree structure can be composed by means of the class area. A necessary property of an area is its physical subaddress (address) relating to its integration in the topology hierarchy. The class instanceArea embodies the instances of areas which contain an address and are linked to entities. Fourth, the categorization of the network in functional terms is covered by the view functional. The structuring element is the group, but unlike the other three views, instances of groups

(`instanceGroup`) refer to a datapoint and not to entities. In order to provide functional access to the network already at the group level, a reference to a datapoint (`function`) has to be embedded in a group. In addition, the attribute `connector` of type `enumConnector` describes the type of participation of a datapoint in a group, i.e. it is a sink of the group (`receive`) or a source (`send`).

Besides these already mentioned views and lists, the network comprises the `references`. This class contains a list of `reference`. Thus, textual information can be stored consistently and free of redundancy. Other classes can have links to a reference like the manufacturer information of an entity.

To avoid restrictions to the design process of a BA network, this metamodel refrains from creating additional OCL constraints. Therefore, a high level of abstract and universal modeling is provided.

3.3.3 Model creation

The BAS metamodel on layer M_2 of OMG's modeling stack defines the modeling language for the creation of BAS models on the next lower level. This section shows a few examples how the BAS metamodel can be used to build different models in order to map an entire BA network. Before a network can be successfully modeled, the relevant meta information has to be constructed. A real network needs elements like *units*, *data types*, *parameter types* or *enumerations*. Appropriate units and enumerations support the implementation of complex and extensive types.

First, Listing 3.3 illustrates an XMI serialized model of a BAS unit which is also used in the implementation and evaluation part of this thesis. This textual representation facilitates a standardized information exchange while graphical visualizations offer a much easier interface for editing such models. Units are multilingual elements that contain an optional dimension expressed in SI units. The stated example shows a unit for degree Celsius values with the SI unit *Kelvin* and an offset value of -273.15 refering to the basic unit. Moreover, the unit has a German translation for the attribute `name`. The translation tag does not show an attribute for the language and the translated attribute because *de_DE* and *name* are the default values of their particular enumeration. Thus, the default value of an enumeration is the topmost literal. The namespace reference `http://auto.tuwien.ac.at/bas` links the model to the BAS metamodel.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <bas:unit xmlns:bas="http://auto.tuwien.ac.at/bas" id="celsius"
    name="temperature (°C)" description="" symbol="°C" offset="-273.15">
3   <translation value="Temperatur (°C)"/>
4   <dimension K="1"/>
5 </bas:unit>
```

Listing 3.3: BAS unit model

Second, an example of an enumeration is examined in Listing 3.4. It is important to distinguish between an enumeration of the metamodel and an enumeration on the model layer which

is based on the metamodel's class `enumeration`. This example describes the possible types of a building part, i.e. each literal represents a permitted characteristic. Alongside, every literal has a unique integer value given in the correspondent attribute.

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <bas:enumeration xmlns:bas="http://auto.tuwien.ac.at/bas" id="part"
       name="Part">
3    <literal id="building" name="Building" value="0"/>
4    <literal id="buildingpart" name="Building Part" value="1"/>
5    <literal id="floor" name="Floor" value="2"/>
6    <literal id="stairway" name="Stairway" value="3"/>
7    <literal id="room" name="Room" value="4"/>
8    <literal id="corridor" name="Corridor" value="5"/>
9    <literal id="distributionboard" name="Distribution Board" value="6"/>
10 </bas:enumeration>
```

Listing 3.4: BAS enumeration model

Third, Listing 3.5 shows a data type example. Here, the KNX datapoint type *DPST-9-1* for temperature values is modeled in terms of the BAS metamodel. As outlined in the previous section, data types are hierarchically organized, and therefore this example type has the KNX parent type *DPT-9* which is located in a separate model in the file `dpt_9.bas`. As the type is used for temperature values, only one property of the type `numeric` is needed to cover the necessary functionality. The already presented Celsius unit is linked with this property via the attribute `unit`. Additionally, the range of the value and a proper name can be modeled.

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <bas:type xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:bas="http://auto.tuwien.ac.at/bas" id="DPST-9-1" name="Value Temp"
       parent="dpt_9.bas#DPT-9">
3    <property xsi:type="bas:numeric" name="value" min="-273.0" max="670760.0"
         unit="../../library/unit/celsius.bas#celsius"/>
4  </bas:type>
```

Listing 3.5: BAS type model

Finally, a network can be built using the evaluated metamodel and the introduced models for meta information. Figure 3.9 visualizes the BA network *Office* with its child nodes for the views, the list of entities and the list of datapoints. The functional view is exemplarily expanded to show the feasible construction of groups and subgroups. Entity instances can be seen as leaves of the group *Light on/off*. Subsequently, Listing 3.6 gives an excerpt of the network model, showing an entity with two associated datapoints (`datapoints`) as XMI serialization.

All in all, the metamodel allows the construction of a huge set of models including libraries for meta information or various networks of BASs.

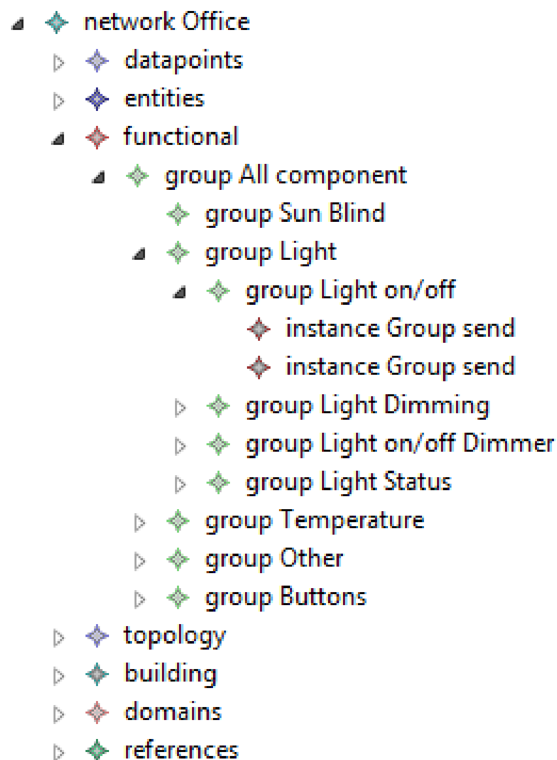

Figure 3.9: BAS network model

```
1    <entity id="P-0341-0_DI-11" name="Temperature Sensor N 258/02"
         description="Productinfo - see file:  2581ab02_tpi_e.pdf"
         orderNumber="5WG1 258-1AB02" manufacturer="#M-0001"
         datapoints="#5F7E_O-0_R-2 #5F7E_O-1_R-3">
2      <translation language="de_DE" attribute="name" value="
           Temperatursensor N 258/02"/>
3      <translation language="de_DE" attribute="description" value="
           Produktinfo - siehe Datei:  1258ab02_tpi.pdf"/>
4    </entity>
```

Listing 3.6: BAS entity model

3.4 Metamodel for oBIX

The concept of MDA introduces a PSM besides the PIM. In this thesis, oBIX is used as target technology for the integration of BASs. Therefore, the metamodel for the platform specific language is designed in accordance with the oBIX concept. The subsequent section describes the modifications of to the oBIX object model in order to achieve the requirements of the already

introduced BAS metamodel. Similar to the metamodel for the PIMs, the correct use of the oBIX metamodel is examined in Section 3.4.2.

3.4.1 Adoption of the oBIX object model

In Section 2.2.2 the oBIX standard and its underlying object model launched by the OASIS were introduced. The approach of this thesis is based on the *Committee Specification Draft 02* of oBIX version 1.1 from December 2013 [49] which is shipped with an XML Schema as machine readable documentation. While integrating this object model in the MDA scheme, a few changes have to be made. For instance, the standard XML data types used for specifying the attributes of the various classes are not applicable anymore. The following itemization describes each implemented modification of the original object model:

Additional attributes for the class `Obj` have to be created. These attributes are `readable`, `transmittable` and `updatable` which correspond to the flags of a datapoint in the BAS metamodel together with the already existing flag `writable`. If these attributes do not exist in the intermediate platform specific metamodel, information will be lost on the model layer between the PIM and the final program code. Although it is possible to add these extra flags as child objects of type `Bool`, they are attached as attributes to be in line with the flag `writable`.

Renaming is necessary for the attribute `href` in `Obj`, as the validation mechanism of the EMF tools is not able to distinguish between the XML attribute and the homonymous attribute of the oBIX object model. Attempts to add an explicit namespace information indicating the particular attribute have failed, and thus it is indispensable to rename `href`. Therefore, the name `uri` is chosen.

Data types in XML are not the same as in the used MDA tool chain. Hence, these types have to be mapped to available simple or complex data types. For instance, the type `duration` has to be replaced by the custom type `EDuration`. More details relating to these data type changes are extensively discussed in Chapter 5.

Naming of classes is slightly modified. `AbsTime` and `RelTime` are written in camel case, but the used oBIX gateway implementation needs `Abstime` and `Reltime` as notation. However, this is just a marginal change and does not influence the functionality of the MDA approach in any case.

Missing attributes are the result of some inconsistency in the oBIX specification. While the class diagram and the XML Schema omit the attribute `tz` for the time zone in `Time` and `Date`, the textual description and former versions of the specification mention this attribute. Similarly, the attribute `of` in class `Ref`, which specifies the type of the objects contained in a referenced list, have been lost in this specification. In order to provide a complete metamodel, these attributes are integrated in the oBIX metamodel.

Besides these adaptions the object model is directly transferred to a metamodel conforming to the MOF meta-metamodel. The result is shown in Figure 3.10. As it can be observed, the similarity to the object model from Section 2.2.2.1 is obvious.

The overall superclass `Obj` defines a set of attributes for its identification and additional description. The attribute `name` in combination with `uri` is used to allocate an object inside a set of oBIX objects. Additionally, the contract respectively the list of contracts is stated in the attribute `is` (see Section 2.2.2.2). Human readable descriptions are given in the attributes `display` and `displayName` corresponding to the attributes `name` and `description` of the BAS metamodel. The meaning and use of further attributes can be found in the oBIX specification [49]. Last but not least, the class `Obj` has a containment association back to itself which enables the creation of a hierarchy of objects.

All other classes are derived from this superclass, and thus inherit its properties. Alongside the previously listed modifications, no further changes are made to these subclasses. Their descriptions can be found in the oBIX specification, as well. In the upper left corner of the oBIX metamodel, the enumeration `status` with all possible status values is modeled. This enumeration is implemented in accordance with the underlying object model. Right next to it, six custom data types are introduced to cover the XML data types from the original object model. As a matter of fact, simple data types from the MDA implementation's tool chain can be used instead, but custom types provide more flexibility for additional functionality or restrictions. For example, the type `EUri` is the counterpart of XML's `anyURI` to store a URI. In most cases, a simple character string would be fine. However, with the custom type, it is possible to check the correctness of a given URI or to guarantee the uniqueness of such an identifier. Similar considerations are the reasons for the remaining types.

Whereas the BAS metamodel is specifically tailored for BASs to model BA networks, the oBIX metamodel defines a very general modeling language. While the former has separate classes for the different views, entities and datapoints, the latter combines them into a small set of common classes. The focus of the oBIX object model respectively its metamodel lies on the uniform integration of heterogeneous information in a gateway technology to provide easy access to the resources. As the name already says, a PSM conforming to an appropriate metamodel focuses on the final target technology.

3.4.2 Metamodel utilization

Within this section, the creation of models based on the oBIX metamodel with the main focus on BA networks is discussed. There are multiple ways of modeling the same information in oBIX. It is the designer's decision which object structure is the most suitable for a certain purpose. In this thesis, the structure of the PIMs based on the BAS metamodel is retained as much as possible. Therefore, the PSMs can be generated much easier. The following listings give brief clippings in XMI of an oBIX model which has been generated from a PIM.

First, the root of a BA network is shown in Listing 3.7. It corresponds to the class `network` of the BAS metamodel and contains the same information as its platform independent counterpart. Due to the limitations of oBIX, attributes become separate objects (e.g. `standard`). Likewise, associations and containments cannot be modeled directly, but need the concept of oBIX references. Such a reference of the type `Ref` is the link to the list of entities (`entities`).

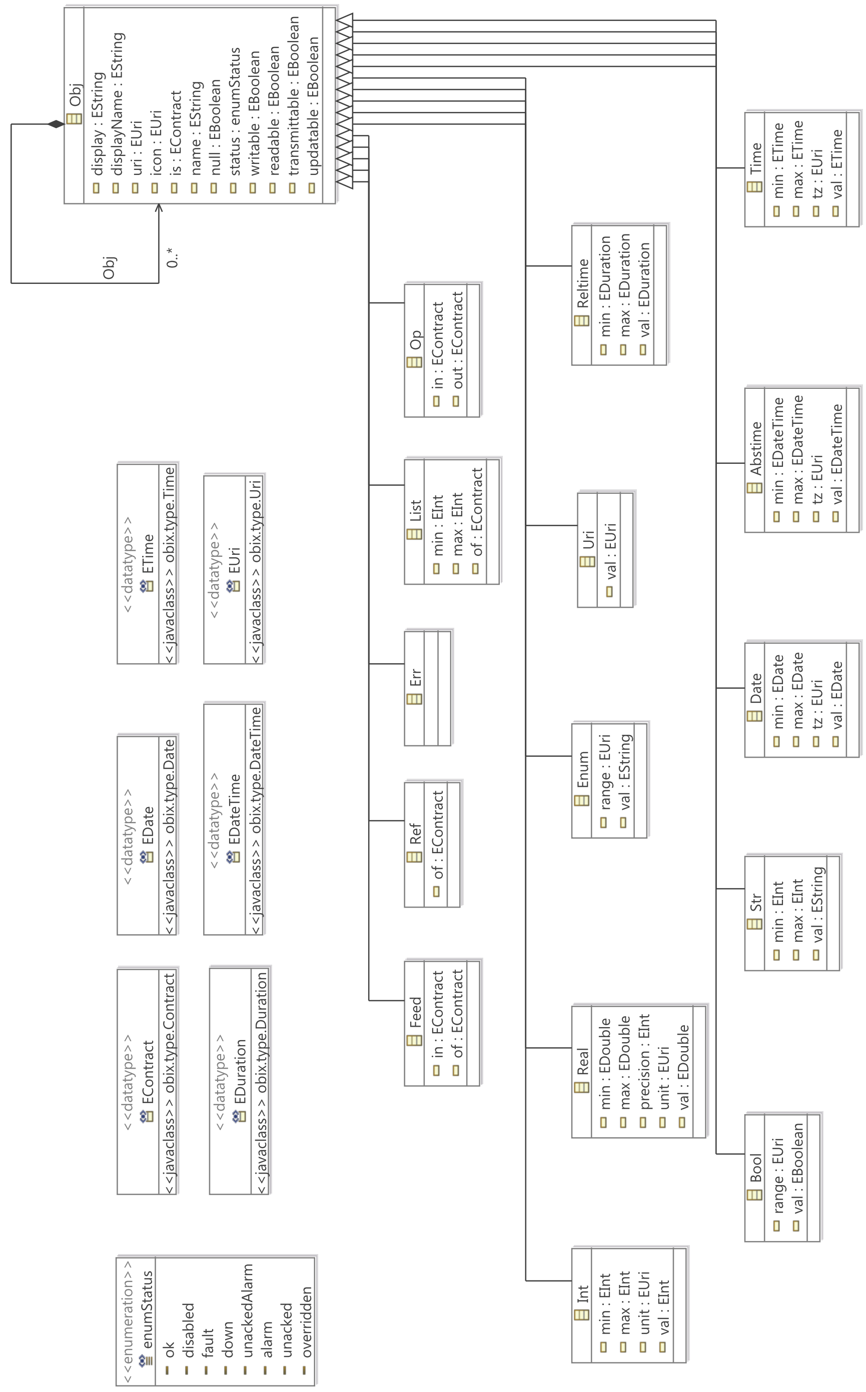

Figure 3.10: oBIX metamodel

By following the relative or absolute URI of a reference (attribute `uri`), the actual object can be addressed.

```
1 <Obj uri="/networks/office" displayName="Office" is="bas:Network"
    name="P-0341">
2   <Obj xsi:type="obix:Enum" uri="standard" name="standard"
      range="/enums/standard" val="knx"/>
3   <Obj xsi:type="obix:Ref" uri="datapoints" is="bas:Datapoints"
      name="datapoints"/>
4   <Obj xsi:type="obix:Ref" uri="entities" is="bas:Entities" name="entities"/>
5   <Obj xsi:type="obix:Ref" uri="functional" is="bas:Functional"
      name="functional"/>
6   <Obj xsi:type="obix:Ref" uri="topology" is="bas:Topology" name="topology"/>
7   <Obj xsi:type="obix:Ref" uri="building" is="bas:Building" name="building"/>
8   <Obj xsi:type="obix:Ref" uri="domains" is="bas:Domains" name="domains"/>
9 </Obj>
```

Listing 3.7: oBIX network model

Second, Listing 3.8 illustrates a model of an entity. Here, the entity is a temperature sensor with two child objects of type `Str` defining the manufacturer name and the order number. Moreover, the entity contains a list of datapoint references indicated by the list's `of` attribute (*obix:ref bas:Datapoint*). These references are of type `Ref` and point at real datapoint objects which reside under the given URI.

```
1 <Obj uri="/networks/office/entities/temperature_sensor_n_258_02/1"
    displayName="Temperature Sensor N 258/02" is="bas:Entity"
    name="P-0341-0_DI-11">
2   <Obj xsi:type="obix:Str" uri="manufacturer" name="manufacturer"
      val="Siemens"/>
3   <Obj xsi:type="obix:Str" uri="orderNumber" name="orderNumber"
      val="5WG1 258-1AB02"/>
4   <Obj xsi:type="obix:List" uri="datapoints" name="datapoints"
      of="obix:ref bas:Datapoint">
5     <Obj xsi:type="obix:Ref" displayName="Temperature, Channel A"
        uri="/networks/office/datapoints/temperature_channel_a/1" is="
        bas:Datapoint" name="P-0341-0_DI-11_M-0001_A-9814-01-5F7E_O-0_R-2"/>
6     <Obj xsi:type="obix:Ref" displayName="Temperature, Channel B"
        uri="/networks/office/datapoints/temperature_channel_b/1" is="
        bas:Datapoint" name="P-0341-0_DI-11_M-0001_A-9814-01-5F7E_O-1_R-3"/>
7   </Obj>
8 </Obj>
```

Listing 3.8: oBIX entity model

Finally, the modeling of a datapoint is examined in Listing 3.9. As already known, datapoints host the device functionality of a BAS. In the PIM, a datapoint has a type which is linked to the datapoint by a reference. On the contrary, the oBIX model includes the properties of the type as child objects directly in the datapoint object (`value`, `encoding`). The `is` attribute shows a

list of contracts representing the inheritance hierarchy of this particular datapoint. At this point, it should be noted that the contract prefix `bas` has nothing in common with the namespace of the BAS metamodel, but should clarify the type of the object as an element of BASs.

```
1 <Obj uri="/networks/office/datapoints/switch_channel_a/1" display="On/Off"
     displayName="Switch, Channel A" is="bas:DPST-1-1 bas:DPT-1 bas:Datapoint"
       name="P-0341-0_DI-3_M-0001_A-9803-03-3F77_O-3_R-4" writable="true"
     transmittable="true" updatable="true">
2   <Obj xsi:type="obix:Bool" uri="value" name="value" null="true"/>
3   <Obj xsi:type="obix:Enum" uri="encoding" range="/encodings/onoff"
       name="encoding" null="true" />
4 </Obj>
```

Listing 3.9: oBIX datapoint model

CHAPTER 4

Transformation process

The following chapter deals with the horizontal interaction of the individual models on the second lowest layer of the modeling stack, M_1 (see Section 3.1). In short, it describes the process from modeling of the BAS to the final generation of executable program code.

4.1 Workflow description

While the previous chapter outlines the static aspects of MDE, the following sections discuss the dynamic issues, i.e. the workflow steps between the particular MDA models and metamodels. The approach considers three distinguishable phases:

1. The ***network modeling*** represents the stage of mapping the BAS to a machine readable model. Whether this step is executed automatically by using data from a BAS engineering tool, or the system integrator builds the model by hand, does not matter. The important thing is the resulting BAS model.

2. The ***model transformation*** translates the PIM into a PSM. In contrast to the network modeling, this step is realized by the support of MDA mechanisms.

3. The step ***code generation*** converts the PSM into executable program code. Again, the generation is not done manually. Instead, the available MDA tools are utilized for implementation and execution.

These three steps are highlighted in Figure 4.1 which depicts a cutout of the overview from the introduction (see Chapter 1). Admittedly, it would be possible to develop a transformation in one step. For instance, the data from the BAS engineering tool are transformed directly into source code. However, this will amongst others dramatically limit transparency, extensibility and reusability. Section 2.1 already pointed out the advantages of a model-driven approach. Moreover, it has already been outlined why a separation between PIM and PSM is highly preferable. Therefore, a transformation process conforming to the MDA initiative is chosen.

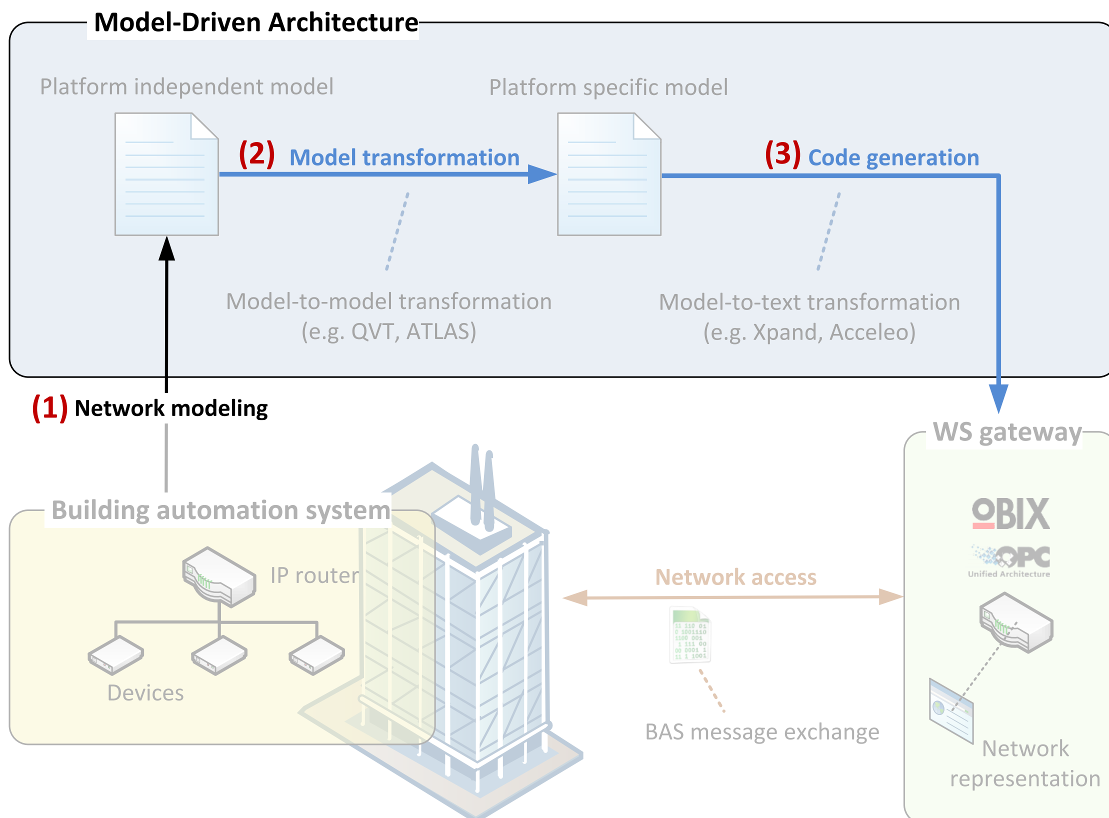

Figure 4.1: Transformation workflow

Each process step describes a set of rules and operations to convert input into desired output. The workflow is located in layer M_1 of the modeling stack. Transformations are performed in horizontal direction between models written in different modeling languages. Due to the taken approach, intermediary results exist before the final program code is generated. These intermediaries are both output of the previous transformation and input for the subsequent process step. Thus, the workflow represents a chain of single steps from the initial BA network to the executable source code. Except for network modeling, the other workflow steps can be realized in accordance with the MDA concept. As soon as the BAS exists as PIM, it can be automatically processed until it is finally integrated into a WS gateway. The network modeling depends on the available BAS engineering tool and the preferences of the system designer.

All in all, the developed MDA approach substantiates Hypothesis 2 (see Chapter 1), as it can be read in detail in the following sections. The BAS as the workflow's input is transformed stepwise. Finally, the network is integrated into a WS gateway. The theoretical concepts in this chapter are supplemented by an implementation in Chapter 5.

4.2 Network modeling

The initial position can be described as a combination of various sensors, actuators and other components that form a BAS. In most cases, access to the system is considerably restricted, hence it is only possible to manage the network with special (proprietary) software tools. Based on this existing, local network, the overall objective of this thesis is to enable remote access via an interfacing technology. The BAS services should be available via a general interface, and modeling of the network is the first step to integrate it into a model-driven approach for further processing. In short, the BAS is the input of this step while its representation as model is the desired output. The transformation can be realized either by manual modeling or by a tool based, automatic generation. Both approaches are discussed in the following two subsections.

4.2.1 Manual approach

The whole MDA process is based on DSLs and the overlying meta-metamodel. For manual modeling of the network, the starting point is the BAS metamodel. The instances of this metamodel, the PIMs, are technology independent representations of the real BAS. A designer's task is the constitution of the network topology and all its connected devices in the form of such a PIM. In this case, this is done manually. Therefore, the designer uses a visual editor or a text editor to create the model objects. This way of network modeling is chosen, if no machine readable data of the underlying network are available.

Within the limits of the metamodel, the designer has a lot of options for mapping the BA network. The focus of the successive BMS largely influences which elements of the actual network are represented in the model. For instance, if a remote BMS does not need information about the partitioning of the building in different parts, recording of the building view in the PIM will not be required. On the other hand, if the BMS provides a service for measuring the energy consumption of a particular domain (e.g. lighting), the relevant view should be implemented. In Section 3.3.3, it was already outlined how modeling by means of the BAS metamodel can be achieved.

4.2.2 Automatic approach

Besides manual implementation of a PIM, an automatic approach via a computer-based transformation can be chosen. In this case, machine readable information about the network structure and its components needs to be available. One way of receiving relevant information is the utilization of data from a BAS engineering tool. If this software offers an export interface, network information can be extracted and processed.

For example, KNX systems can be constructed by means of the Engineering Tool Software 4 (ETS4) [19]. This application exports the BAS data as XML files that conform to a given XML Schema. As already pointed out, this thesis relies on a KNX network to evaluate the developed model-driven approach (see Chapter 6). In this context, an automatic transformation script for network modeling has been built based on the BAS metamodel and the KNX XML Schema. Available information is mapped to a model conforming to the BAS metamodel by the use of XSLT.

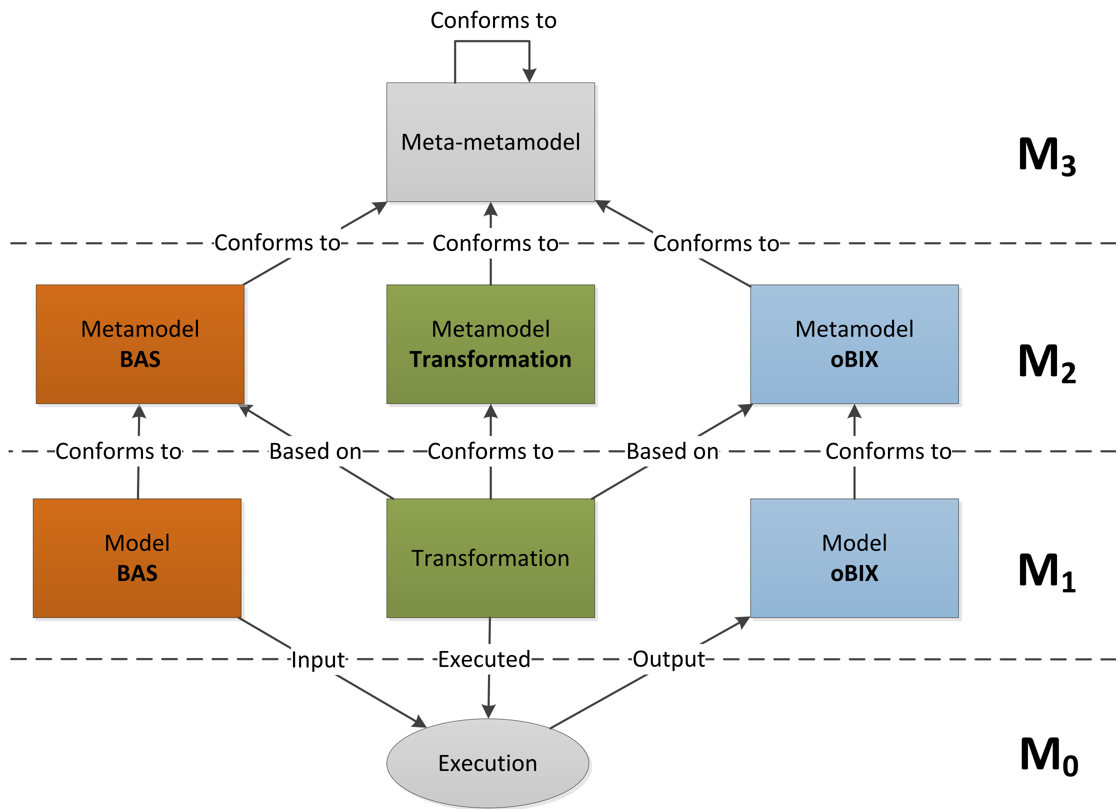

Figure 4.2: Model transformation from BAS to oBIX

It seems that only automatic realization of the network modeling is consistent with Hypothesis 2 which speaks about a fully automated transformation process. However, this first step is only the initial activity while the actual MDA approach starts more or less with the existence of a PIM. Thus, manual modeling of the network does not refute the stated hypothesis.

4.3 Model transformation

The second step in the integration workflow is the transformation from the PIM to the PSM. In terms of MDA this is also called M2M transformation [7]. As already outlined (see Section 3.3), the PIM is the model conforming to the BAS metamodel. On the other hand, the PSM is the model conforming to the oBIX metamodel. According to Jouault et al. [30], Figure 4.2 visualizes the involved process components. In the figure, these are embedded in the well-known modeling stack, whereas the transformation and its metamodel are located between the source (BAS models and metamodel) and the target (oBIX models and metamodel). The execution on layer M_0 has a BAS model as input. The transformation rules of layer M_1, which conform to the transformation metamodel, modify the input data and create an output model. In this workflow, the output model represents a BA network in terms of the oBIX modeling language.

The aim of this process step is the translation of information from a platform independent representation into a technology specific syntax for further processing. MDA enables the opportunity to change the target platform by simply replacing the transformation's output model. For instance, OPC UA can be chosen. Therefore, the transformation must generate an OPC UA model of the BA network. These changes will not affect the original PIM due to the separation of the workflow into distinct phases. In summary, this M2M transformation creates platform specific objects of the underlying BAS by means of appropriate transformation rules.

The following subsections outline the addressed transformation in detail. First, the developed conversions are discussed in a general way (transformation rules) relating to the M_2 layer of the modeling stack that contains the metamodels. Each visualization shows a cutout of the BAS metamodel on the left hand side and the corresponding part of the oBIX metamodel on the right hand side. Second, the concrete transformation of a BAS model into an oBIX model (transformation execution) is presented. The execution utilizes models on the M_1 layer of the stack as input and output whereby the focus is on the actual BA network parts. Therefore, the transformation of library models (e.g. data types, units) and other meta information is mostly omitted in the next two subsections.

4.3.1 Mapping of metamodels

In this section, various rules for converting a BAS model to an oBIX model are shown in terms of the metamodels' language concepts. These rules are aggregated in the item *Transformation* in Figure 4.2. In order to understand the first diagram, a few basic rules for translating elements from the BAS metamodel (source metamodel) to the oBIX metamodel (target metamodel) need to be established:

- Attributes of a class in the source metamodel are converted into child objects of the particular object in the target metamodel, if no adequate attributes in the target class exist.
- Containment associations are often resolved as a list of child objects. The subordinated objects can be either encapsulated in an intermediate `List` object or added directly to the parent object. Nonetheless, exceptions to this rule exist which are discussed later on.
- Standard associations in the source metamodel usually become `Ref` objects in oBIX. In some cases, containments are resolved as references, as well. Due to a better structure of the overall model, the subsidiary objects are located in separate oBIX objects. They are linked to the original parent object by an oBIX reference object.
- Abstract source classes are not directly mapped to target classes. The attributes of such abstract classes are included during transformation of the derived classes that are not abstract.

The first visualization of a transformation rule can be observed in Figure 4.3. The left hand side displays a set of classes of the BAS metamodel as the source of the transformation while the right hand side illustrates the needed classes of the oBIX metamodel as the target of the transformation. Here, the transformation of `element` and `translation` is displayed. As

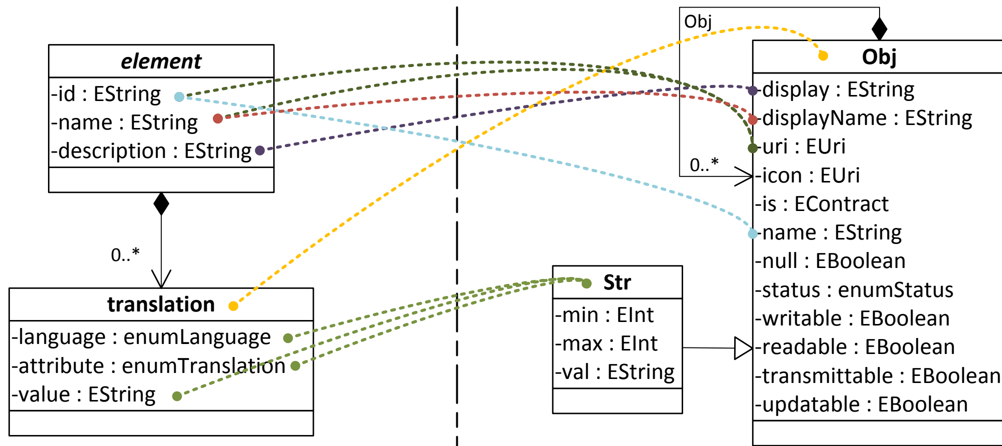

Figure 4.3: Mapping of element and translation

`element` is an abstract class, it is not transformed into an oBIX object. However, its attributes are needed while transforming the derived classes (e.g. datapoints). The dotted arcs demonstrate which part of the source metamodel becomes which part of the target metamodel. Thus, the value of the attribute `id` is mapped to the attribute `name` of `Obj`. Likewise, `name` and `description` are converted to `displayName` and `display`. A special case is the oBIX attribute `uri`. In dependence of the source class, this attribute is composed of `id`, `name` or both attributes. The class `translation` becomes an `Obj` with three child objects of type `Str` due to its three attributes. Generally, child objects are linked via the containment association of `Obj` to the parental object. At this point it has to be noted, that the value of attribute `is` comes from the particular source class and is not considered explicitly in the shown figures. For instance, the source class `translation` leads to the contract *bas:Translation* in the `is` attribute.

Translations are one of the exceptions (cf. basic rule for containment associations above) as they are not linked with the superordinated element in the oBIX model. Hence, they are integrated completely autonomously. Subsequent transformation steps use the translation's value of the `name` attribute to establish a link between an object and its translations.

Figure 4.4 represents the mapping of the BAS' main element, the `network`, and the associations to the network's various views. A network is converted to an `Obj`. The attributes `display`, `displayName` and `name` are filled by the attributes of `element` as `network` is derived from `element`. The attribute `standard` becomes an `Enum` object in oBIX at which the attribute `range` will refer to an enumeration corresponding to the enumeration of the BAS metamodel (`enumStandard`). All associated views are added to the network object as `Ref` objects. The class `references` is not included in the transformation, and therefore it is omitted in the figure. The `uri` attributes of the `Ref` objects refer to the actual view objects in the oBIX model. Consequently, a network has seven child elements, i.e. a `Ref` for each view and an additional `Enum` to store the technology of the BA network. The `Ref` objects cannot be encapsulated in one single list because they refer to different types (contracts), and an integration into separate lists is not necessary as the associations are only of cardinality 1:0..1. If a view

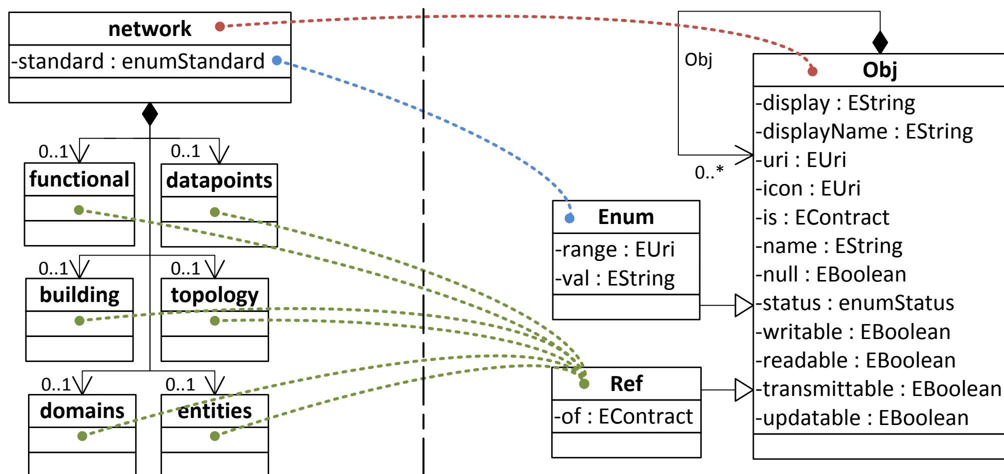

Figure 4.4: Mapping of network

does not exist in the network model, it will not be generated during the transformation.

One of the mentioned views is the plain list of available datapoints. Within the oBIX object of the network, the class `datapoints` is mapped to the class `Ref`. In addition, the list of datapoints exists as a separate oBIX object in the form of a `List` object. The `of` attribute's value is set to *obix:ref bas:Datapoint* because this list contains references to datapoints. Figure 4.5 illustrates this mapping. Similarly, the transformation of the `entities` list uses the same pattern. Entities become a `List` object containing one `Ref` object per entity. Thus, this transformation part is not displayed separately.

The views building, topology, domains and functional possess a very similar structure. Especially, the three former concepts differ only in the naming of classes and some attributes within these classes. In contrast, the functional view does not refer to entities on the instance level, but links the groups and instances with datapoints. Figure 4.6 displays the considerations taken into account when transforming the functional view to its oBIX equivalent.

First, the classes `functional`, `group` and `instanceGroup` are mapped to the standard class `Obj`. Then, the two containment associations to create a tree structure of groups and subgroups as well as the containment association for the instances are transformed to child objects of type `List`. In addition, a group gets a child object for the attribute `address` which is mapped to `Int`. Likewise, `connector` becomes an `Enum` with a URI (`range`) to the oBIX equivalent of the enumeration `enumConnector`. Both a group and a group's instances refer to a datapoint. These associations are mapped by means of a `Ref` object. The datapoints are located somewhere else in the model, but the references provide a link to them. The remaining views are not illustrated as the mapping is similar. Links to the entities are also realized by a `Ref` object. Moreover, the fundamental structure, i.e. a hierarchy of different levels of building parts, areas or domains with instances on each level, is unchanged.

Generally, entities are containers for datapoints. If a container is physically existent, it will be usually a device with an order number and a manufacturer. In Figure 4.7, the mapping of

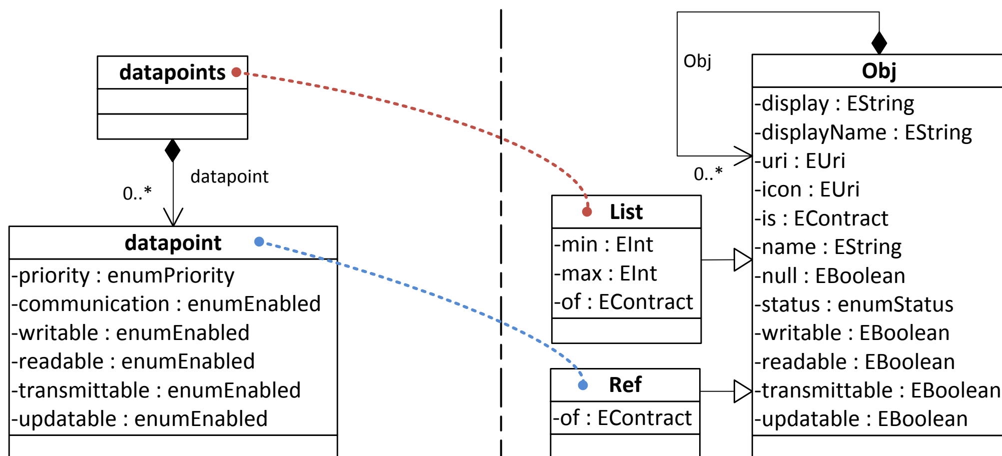

Figure 4.5: Mapping of datapoints view

such an entity to oBIX classes is visualized. First, the `entity` class becomes an `Obj`. Next, the attribute `orderNumber` and the manufacturer name, which is stored in an instance of the class `reference` (accessible via the association `manufacturer`), are converted to the class `Str`. The references are used as redundancy-free storage for repeatedly occurring texts. Instead of being directly transformed to a specific platform technology, the particular texts are inserted when resolving an association to such a reference during the mapping process. Hence, the PSM is not necessarily free of redundancy regarding text values. Finally, a `Ref` object is created for each referenced datapoint. The set of references is integrated into a `List`.

Mapping of a datapoint is one of the most relevant parts in this context. The behavior of a device is defined by its datapoints. Therefore, a comprehensive yet simple representation for a datapoint in the PSM has to be found. The transformation rules applied in this thesis are depicted in Figure 4.8. As summarized in Section 3.3, a datapoint in the BAS metamodel has one unique type. Initially the class `datapoint` is converted into an `Obj`. The values of the flags `writable`, `readable`, `transmittable` and `updatable` of the enumeration type `enumEnabled` are converted into the corresponding boolean attributes of the class `Obj`. In the context of this thesis, the attributes `priority` and `communication` are not used in the oBIX model although they can provide beneficial information for other applications.

Now, the properties of the assigned type must be added to this oBIX datapoint object. The abstract class `property` has an attribute `name` which becomes the `name` of the datapoint's child object. The types of these objects are defined by the respective property class. For instance, an `operation` is mapped to the `Op` class in which the input and output parameters are derived from the associations `in` and `out`. The oBIX equivalent for the class `bool` is the class `Bool`. `Encoding` is mapped to `Enum` in which the `range` is determined by the associated `enumeration`. While `integers` are converted to `Int`, floating-point numbers (`numeric`) become an instance of class `Real`. The attribute `unit` of these classes is set to the referenced unit. As the other property classes are not used within this thesis, they are omitted in the figure. However, Table 4.1 summarizes the mapping of all property classes.

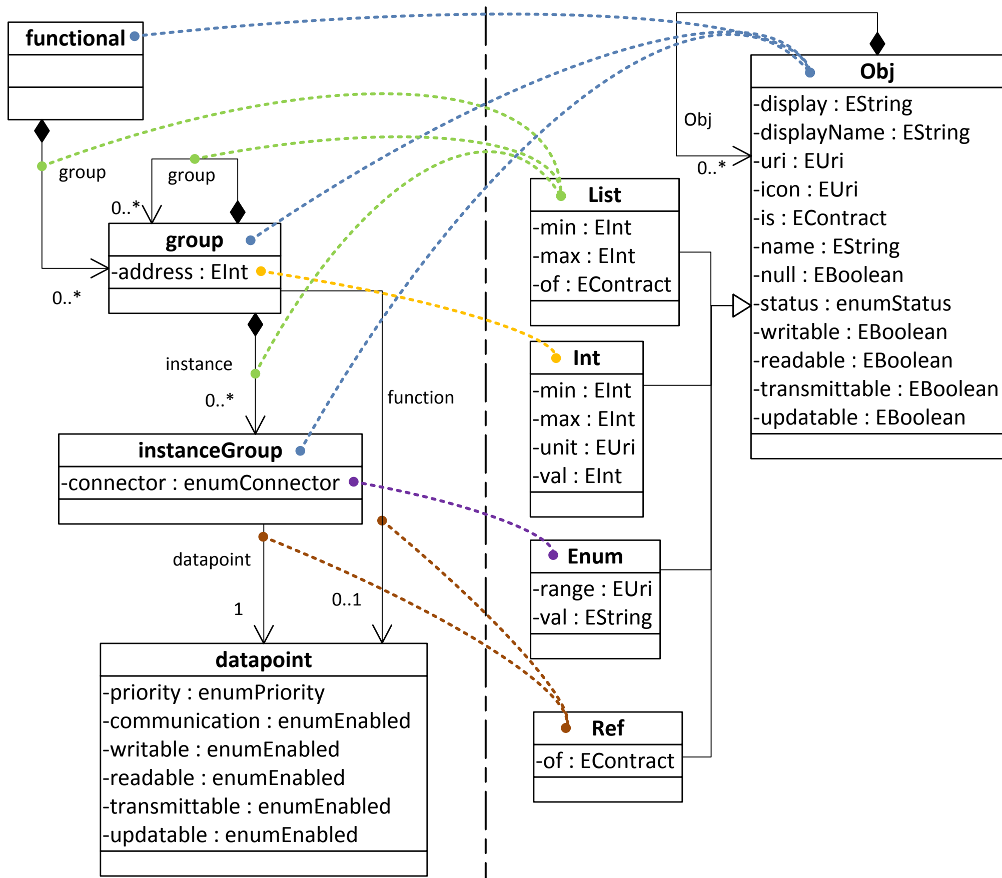

Figure 4.6: Mapping of functional view

4.3.2 Mapping of models

The general transformation rules of the metamodel classes in the previous subsection can be applied to any BAS model on the M_1 layer of the modeling stack. Input models are instances of the BAS metamodel. They represent a real BA network and are expressed in terms of the modeling language defined by the metamodel. The output of this transformation is an instance of the oBIX metamodel. General and platform independent information of the BAS model is converted into platform specific terminology of oBIX. The transformation of a concrete model is located in *Execution* of layer M_0 (see Figure 4.2) in which the transformation model (i.e. the rules) is executed.

Similar to the last subsection, the focus is on the transformation of a BA network (e.g. datapoints, views) and not necessarily on its meta information (e.g. types, units). For simplification, only a few examples are given to demonstrate the application of the transformation rules. The examples are visualized similarly to the previous ones of Section 4.3.1. However, the upper half of the examples shows pieces of a BAS model whereas the lower half of the diagrams illustrates the resulting oBIX elements as output of the transformation execution. Thus, the horizontal line

Figure 4.7: Mapping of entity

can be seen as execution engine between input and output model.

First, the transformation of a network is examined. The example network *Office* is used for this purpose as it represents a small KNX network. The evaluation in Chapter 6 discusses this KNX network in detail because it is also part of the thesis' proof of concept implementation. The various conversion steps can be seen in Figure 4.9.

In the BAS modeling language, a network is an instance of the class `network` and has at most seven child elements. The class `network` is mapped to the class `Obj`. Thus, the network instance becomes an oBIX `Obj`. The child elements are converted into subordinated `Ref` elements, but the element `references` is not converted into the oBIX model. Each `Ref` object points to the instance of the particular view or list, e.g. the reference `entities` refers to the actual list of entities. The table of properties, which is connected with the network instance by the blue arrow, lists all available attributes of the *Office* network. While the attributes `description`, `name` and `id` are inherited from the class `element`, the property `standard` is defined in the class `network`. Attributes of BAS classes with corresponding attributes in the oBIX counterpart are mapped directly. For instance, the value *Office* of `name` is transferred

property
-name : EString

datapoint
-priority : enumPriority
-communication : enumEnabled
-writable : enumEnabled
-readable : enumEnabled
-transmittable : enumEnabled
-updatable : enumEnabled

0..*
1
type
in
out
0..1
operation
encoding
enumeration
enumeration
integer
-min : EInt
-max : EInt
unit
-symbol : EString
-offset : EFloat
-scale : EFloat
unit
unit
numeric
-min : EDouble
-max : EDouble
bool

Obj
0..*
Obj
-display : EString
-displayName : EString
-uri : EUri
-icon : EUri
-is : EContract
-name : EString
-null : EBoolean
-status : enumStatus
-writable : EBoolean
-readable : EBoolean
-transmittable : EBoolean
-updatable : EBoolean

Op
-in : EContract
-out : EContract

Enum
-range : EUri
-val : EString

Int
-min : EInt
-max : EInt
-unit : EUri
-val : EInt

Real
-min : EDouble
-max : EDouble
-precision : EInt
-unit : EUri
-val : EDouble

Bool
-range : EUri
-val : EBoolean

Figure 4.8: Mapping of datapoint

BAS metamodel (source)	oBIX metamodel (target)
bool	Bool
date	Date
datetime	Abstime
duration	Reltime
encoding	Enum
integer	Int
numeric	Real
operation	Op
text	Str
time	Time

Table 4.1: Mapping of type properties

to the attribute `displayName`. This can be seen in the property table of the oBIX network object. In contrast, the attribute `standard` has not any appropriate oBIX attribute for a direct conversion. Therefore, a new object of type `Enum` has to be created as a child of the network object. This enumeration object gets the same name as the attribute in the BAS modeling language. The value of the network's attribute is transferred to the attribute `val` of the `Enum` object. Additionally, the `range` is set to a separate oBIX object which defines the allowed values of the enumeration. The generated URI is set relatively to the parental network object. A relative URI is indicated by the missing leading slash. The value *bas:Network* has been set as a suitable contract in the `is` attribute for the class `network`. For unique identification and discovery, the URI value (attribute `uri`) is generated from the network name and the location of all network objects within an oBIX gateway. `Writable` is set to *false* in all created objects. Writable datapoints and their properties are the only objects where `writable` can be set to *true*. For a better readability, some attributes (e.g. `readable`, `icon`) are left out in the tables of oBIX object properties. This is true for both Figures 4.9 and 4.10.

Transforming a datapoint from the PIM into the PSM is more complex and comprises a greater set of resulting objects containing translations, references in the datapoints view and the datapoint object with its type properties. All the involved input and output objects and their interconnection are visualized in Figure 4.10.

In the upper left corner, an instance of the class `datapoint` describing a channel of a switching actuator is illustrated. Each datapoint has a set of attributes listed in the property table at the top of the figure. Their mapping has already been described. The attribute `type` is linked to the KNX datapoint type *Switch* which contains a boolean flag (`value`) and an encoding (`encoding`) for the semantic interpretation of this flag. The subordinated elements of the datapoint are translations of its name and its description in multiple languages. As an example, the first translation is shown by the property table just below the datapoint instance. The oBIX translations of multilingual elements (e.g. datapoint, entity, area) are encapsulated in a list. Each translation in this list has a unique number and three child elements of type `Str`. These `Str` objects contain the language, the translated attribute and the translation text corresponding to the

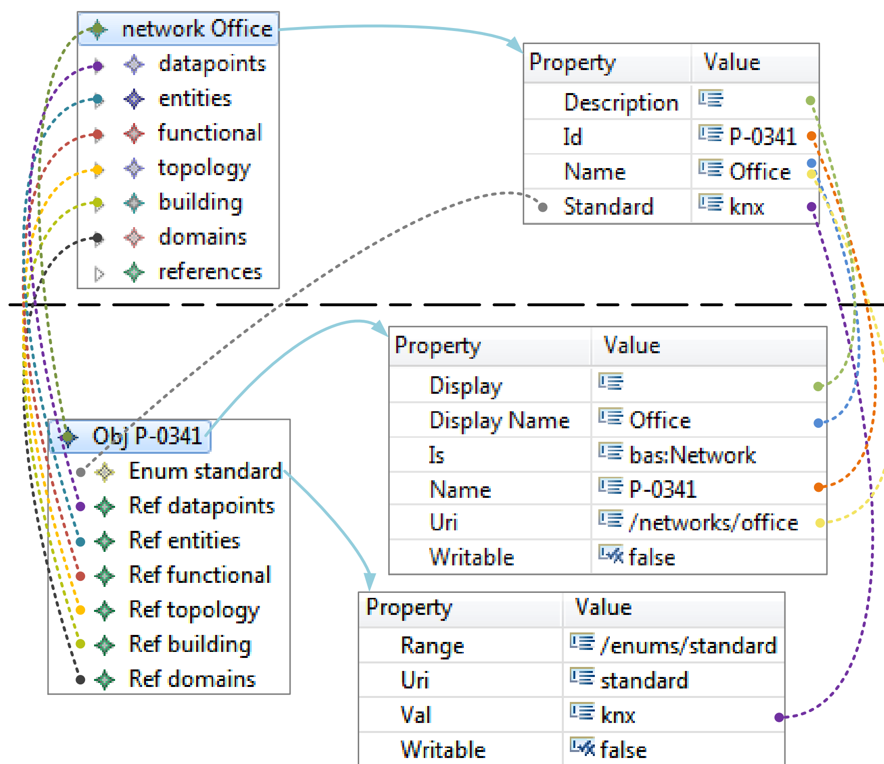

Figure 4.9: Mapping of example network

translation in the BAS model. The results are located on the left hand side of the figure.

Besides the translation objects, the transformation of a datapoint leads to an oBIX object of type `Ref` in the list of datapoints which is again referenced by a `Ref` object in the network object. The datapoint reference obtains its attribute values from the source datapoint. Again, the URI is relative as the actual datapoint is located in the same branch of the oBIX gateway's object tree. The real datapoint is created by transforming the source datapoint into an instance of the class `Obj`. The attribute values are procured from the source properties, e.g. the `name` *Switch, Channel A* becomes the `displayName`. Here, the URI is absolute to enable for direct addressing of this object in the gateway. The `is` attribute contains the full list of superordinated contracts. For adoption of the datapoint functionality, the type properties from the BAS model are attached as distinct oBIX objects within the datapoint object.

4.4 Code generation

The last step in the transformation process is the generation of executable program code. This concluding element in the transformation chain is called M2T transformation [7], and is not only limited to code but also includes the generation of documentation or test cases. Within the workflow cycle, code generation is the third phase after modeling the network by creating a PIM and

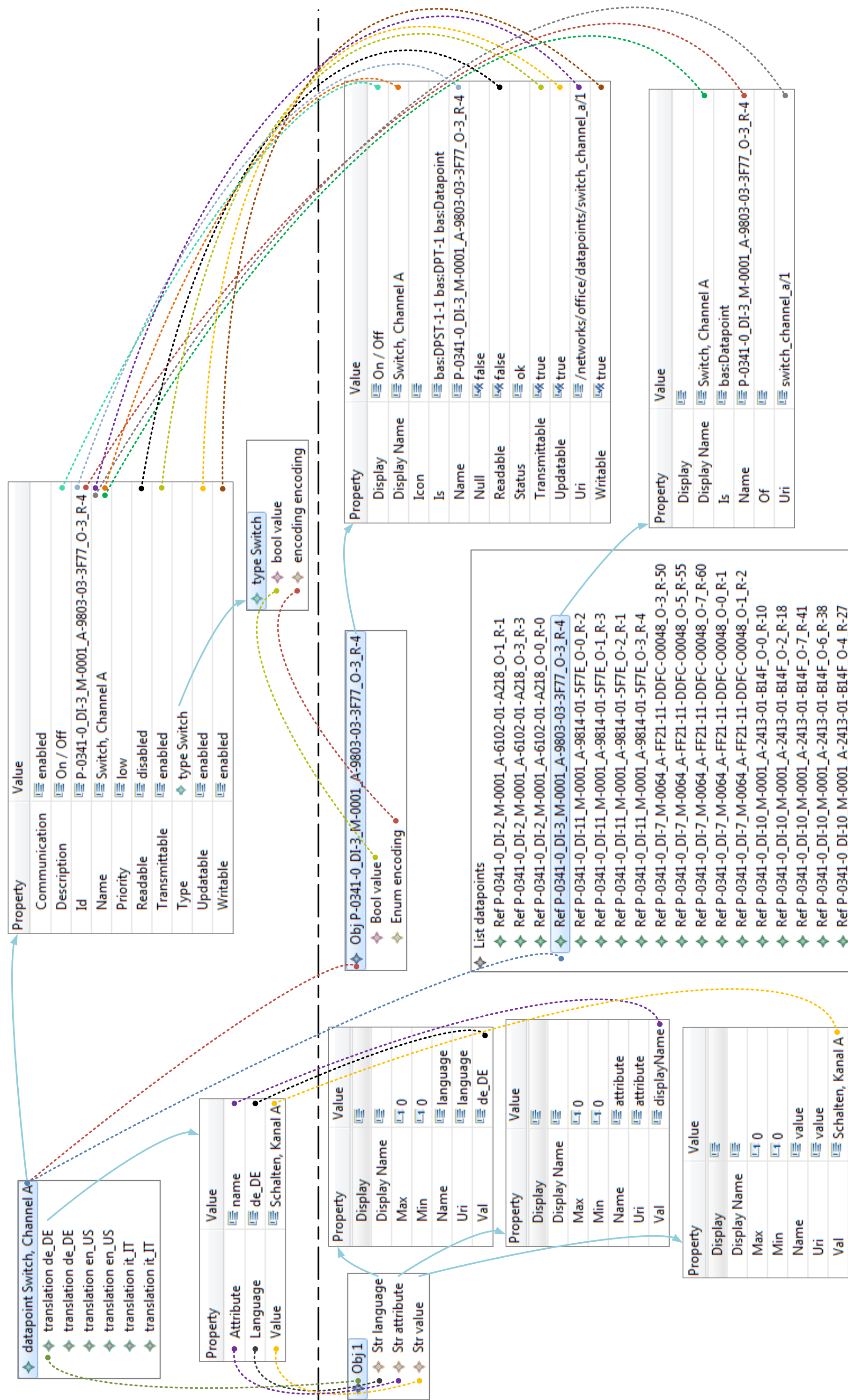

Figure 4.10: Mapping of example datapoint

converting this PIM into a PSM by means of an M2M transformation. While network modeling and the PIM are independent of the subsequent integration technology, the PSM determines the target platform. Last but not least, the code generation assigns the final implementation of the used platform. In this thesis, a Java implementation of the oBIX standard is used for the proof of concept implementation. Hence, this last workflow step generates Java source code by analyzing the information given in the PSM. More details on the used Java implementation can be found in Chapter 5.

The input for code generation is the PSM. Although it conforms to a modeling language which is already aligned with a concrete platform, it is still quite independent regarding the actual implementation of the platform. Thus, the oBIX metamodel provides a language for creating platform specific but implementation independent models. In MDA, descriptions of the platform concepts, i.e. its parts and services, are called platform models [51]. During code generation, the information of the BAS stored in the PSM is transformed into an executable program considering the oBIX platform model. The source code is the output of the code generation which can be simply integrated in the target framework. For the underlying work, a separate Java file is generated and integrated into the oBIX gateway implementation. If the gateway contains all necessary libraries and packages, no further modifications will have to be made in this output file. All in all, the code generation completes the model-driven workflow to integrate a BAS into a WS gateway technology.

Czarnecki and Helsen present two approaches for M2T transformations [15]. Throughout this thesis, the template-based approach is used to transform models into code. Here, the transformation engine, which executes the transformation, needs two input parameters [12]:

- The first parameter, the *source model*, contains the necessary information for the underlying BAS to provide a complete integration of the system. According to the chain of transformation steps, the input for code generation is the output of the model transformation which is the PSM.

- The second parameter is a *template* which is a raw version of the final source code. It predominantly consists of the target text with additional meta code to access data of the source model [15]. For this purpose, the metamodel of the PSM must be linked with the template. Thus, the transformation engine knows the structure of the source model.

The input parameters and the resulting output can be seen in Figure 4.11 which briefly shows the procedure of code generation in the form of an MDA M2T transformation [12]. The meta code in the template is replaced by information from the source model during the execution, while the target text is directly transferred to the output code. Similar to the other models (PIM, PSM), the generated code resides on layer M_1 of the modeling stack. Thus, it is a model conforming to a Java metamodel. The corresponding M_0 element is the execution of the generated program code. In the following paragraphs, the general concept of transforming models into code is discussed. Section 5.2.4 presents the implemented template in more detail.

The aim of code generation is the nondissipative transformation of information from the oBIX model into Java source code. Therefore, each object in the PSM is traversed and converted into a set of Java commands. Within this thesis, each BA network model becomes a single

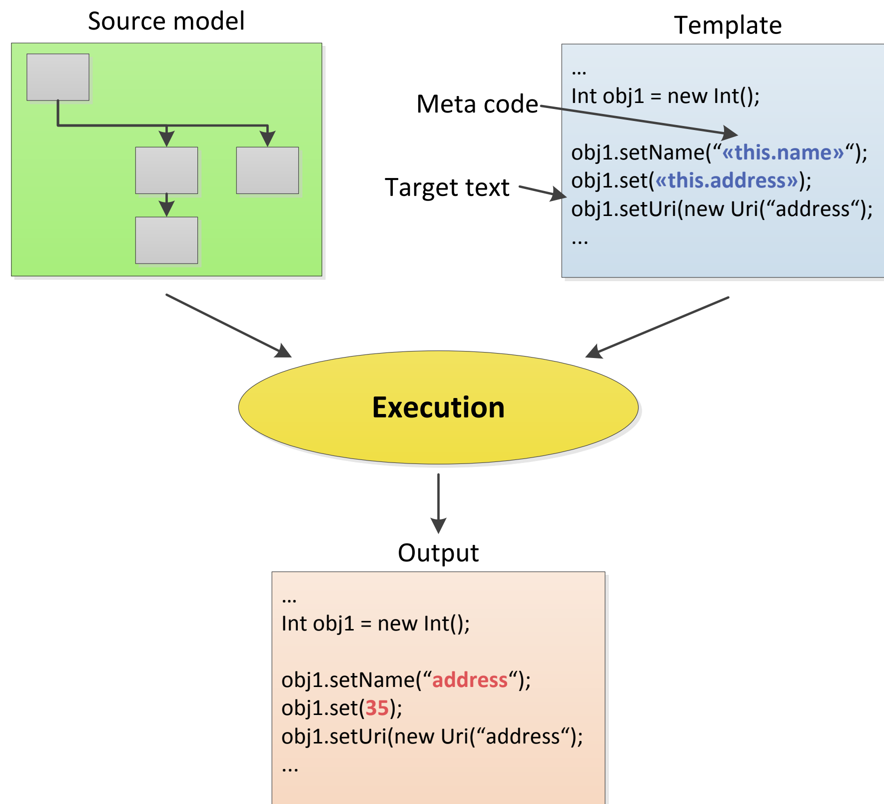

Figure 4.11: Code generation procedure

Java class which contains all the relevant code to instantiate a representation of the BAS in an oBIX gateway. The necessary oBIX classes (e.g. `Obj`, `Enum`, `Int`) are imported from a library implementation, the so called oBIX toolkit [50]. Beginning with the network object of the model, the elements are scanned top down. For this purpose, a template definition for an oBIX `Obj` is created within the transformation template. This definition contains all target text and meta code to instantiate a single oBIX object which is either of type `Obj` or of any other derived type. A recursive call of this definition enables the traversation of the whole model. This way, child elements can be easily added to their particular parent objects. According to the model element type, the corresponding Java implementation of this oBIX class is utilized. For instance, the `address` of type `Int` becomes an instance of the homonymous class contained in the oBIX toolkit. This generic procedure triggers a nearly straightforward set of generated Java instructions embedded in a single class.

In the lower half of Figure 4.9, the root element of a BA network has been shown with respect to the oBIX metamodel. Listing 4.1 shows the program code which is generated during the transformation of these oBIX elements in the last step of the transformation workflow. Irrelevant lines are omitted in this example. The aim of this listing is to enforce a better understanding of the resulting code and its structure.

```
// create generic object
Obj _13912462018330 = new Obj();

// init object
_13912462018330.setName("P-0341");
_13912462018330.setDisplayName("Office");
_13912462018330.setIs(new Contract("bas:Network"));

// add existing translations
if (translations.containsKey("P-0341"))
{
  for (ArrayList<String> entry : translations.get("P-0341"))
  {
    _13912462018330.addTranslation(entry.get(0), entry.get(1), entry.get(2));
  }
}
...

// add as hidden object to object broker
objectBroker.addObj(_13912462018330, true);

// create generic object
Enum _13912462018411 = new Enum();

// init object
_13912462018411.setName("standard");
...

// set value
_13912462018411.set("knx");
_13912462018411.setRange(new Uri("/enums/standard"));

// add to parent (containment)
_13912462018330.add(_13912462018411);

// create generic object
Ref _13912462018412 = new Ref();

// init object
_13912462018412.setName("datapoints");
...

// add to parent (containment)
_13912462018330.add(_13912462018412);
```

Listing 4.1: Example of generated source code

The first four instructions result from the execution of the template snippet given in Listing 4.2. Herein, an `Obj` is created and initialized. The attribute values are taken from the source model (i.e. the PSM) by resolving the meta code given in the template. The command in line 20 (see Listing 4.1) adds the new object to the so called object broker which contains a set of

all registered oBIX objects. Subsequently, the `Enum` object for the definition of the BAS's technology is created. This instance is added to the parental network object in line 34. The example concludes with the implementation of the reference to the datapoints list.

```
// create generic object
«this.metaType.name.split(":").get(2)» «name» = new «this.metaType.name.split
    (":").get(2)»();

// init object
«name».setName("«this.name»");
«IF this.displayName != null»
  «name».setDisplayName("«this.displayName»");
«ENDIF»
«IF this.display != null»
  «name».setDisplay("«this.display»");
«ENDIF»
«IF this.is != null»
  «name».setIs(new Contract("«this.is.toString()»"));
«ENDIF»
```

Listing 4.2: M2T template snippet

Finally, there are exceptions from the almost generic instantiation of oBIX classes. First, the translations are stored in a separate Java map at the beginning of the code generation. They are not added to the set of objects, but they add multilingual texts to the created objects (cf. lines 10–16 in Figure 4.1). Second, datapoints are not implemented by generic oBIX objects. As it is necessary that the datapoints can interact with the underlying BAS, they have to be implemented separately in individual classes. The functionality of sending and receiving messages within the BAS technology is integrated in these classes. During code generation, these datapoint classes are instantiated instead of building up the datapoint in terms of standard oBIX objects.

CHAPTER 5

Implementation

In addition to the general and mostly theoretical considerations given in the previous chapters, the following sections discuss the implementation of a model-driven approach for the integration of BASs into a Web services-based technology. Besides the environment configuration and the realization of the approach, the overall deployment is presented in this chapter.

5.1 Configuration

It is necessary to prepare an appropriate development basis before starting the implementation of metamodels and transformations within this MDA approach. The target technology is a gateway server which implements the OASIS Open Building Information Exchange standard in the version 1.1. This server is realized as a Java application although the programming language is not determined in the oBIX standard.

As a result, a sufficient development environment has to be chosen. Admittedly, the preceding MDA workflow from modeling the BA network to the code generation does not need to be implemented with Java tools. It is sufficient, if the final program code fits to the Java implementation of the oBIX gateway. However, the use of already available Java projects and tools coming with the MDA initiative is recommended. This way, cuts between the development technology and the runtime technology are circumvented, i.e. Java can be used to generate source code of the same programming language that runs on a Java platform. Hence, Java can be seen as solid basis for all development and deployment activities undertaken in this implementation process.

The utilized tools and projects are described in the next two subsections. At first, the installed development environment is examined. Furthermore, the setting of the environment and its associated components that are required for this model-driven approach is presented. It is not only about the types of components, but also about how these components are configured. Second, the gateway implementation is surveyed. As already mentioned, an oBIX implementation is running on a Java platform. In this context, its functionality is discussed. On the one hand, the interaction with the BA network and on the other hand with the remote users.

5.1.1 Development environment

Initially, an adequate development environment has to be selected before implementation is started. It is better to establish as few different tools as possible in order to avoid numerous tool changes while implementing the model-driven approach and the constitutive transformation process. Additionally, it is essential to consider that such a development environment is faced with a lot of different requirements:

Metamodeling establishes the basis for the MDA approach. The development environment must support the creation of domain specific modeling languages as metamodels. These metamodels need to conform to a unique meta-metamodel.

Modeling is the subsequent process of creating a representation of a real system by means of the defined modeling languages. Tools should provide intuitive editors for building models that conform to metamodels.

Validation of the models against the requirements of the metamodels is inevitable. Not only a syntactical inspection has to check the grammatically correct usage of the metamodel's language concepts, but also the compliance of semantic contraints needs to be verified in this task.

Coding of additional Java libraries and standalone auxiliary applications is mandatory to install the postulated, fully automatic workflow.

Model transformation between models of different abstraction levels and languages is an essential part of the MDA initiative.

Code generation is the final part of a model-driven workflow. Thus, an adequate tool support is necessary to fulfil this requirement.

Execution of the implemented transformations or applications must be feasible within the development environment. Hence, the environment should not only provide various editors, but also facilitates an appropriate execution framework.

Based on these requirements, the integrated development environment (IDE) *Eclipse* has been chosen as the best alternative due to its extensible plug-in system [1]. It is possible to combine all previously mentioned tasks in this Java development environment by using a set of additional packages and plug-ins. The general basis is formed by the latest stable Eclipse release 4.3.1 (*Kepler SR1*). The Eclipse Foundation offers different package solutions besides the plain standard version of their IDE. In this work, the solution *Eclipse Modeling Tools* containing a set of tools of the Eclipse Modeling Project is used [18].

In the following paragraphs, the used plug-ins are presented. Figure 5.1 visualizes these components on top of the underlying plain development environment. The Eclipse Modeling Tools solution already includes some of the packages while others have to be installed separately. This can be done by using the native software installer of the Eclipse platform. Another way is to download and integrate these packages with the included installer for modeling components.

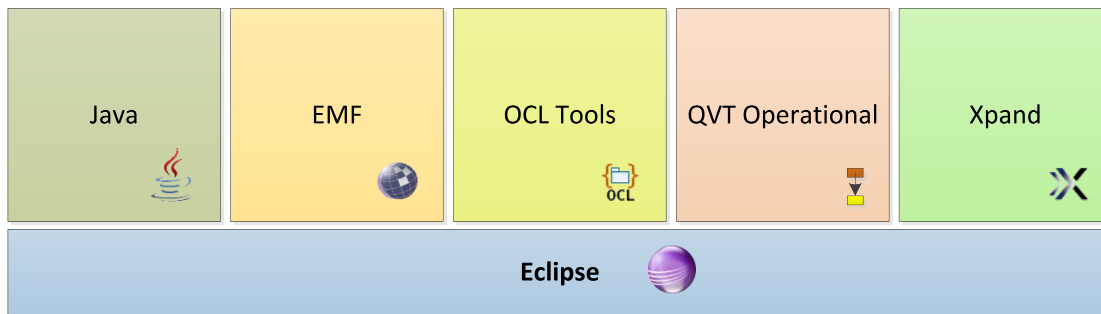

Figure 5.1: Development environment and components

The latter lists all relevant extensions for modeling, transformation or code generation grouped in different categories (e.g. *Concrete Syntax Development* or *Runtime and Tools*).

Although Eclipse has been intended as Java IDE, its focus is now on establishing an open development platform to administer different kinds of software and applications [1]. For the realization of the just introduced requirements, five components are used on top of the generic Eclipse environment.

First of all, the Java Development Kit (JDK) in the version 1.7.0_25 is integrated in order to enable the development of standard *Java* applications. Consequently, it is possible to create individual library packages which are used in the subsequent MDA specific tools. It is not necessary to install additional extensions for the activation of Java development within Eclipse. The installed JDK is simply linked with the downloaded Eclipse version.

The next component is the *EMF*. The core of this package contains the meta-metamodel Ecore which conforms to the MOF standard. As mentioned in previous chapters (see Chapter 3), Ecore is used to describe specific metamodels and modeling languages. Furthermore, EMF consists of classes enabling for building editors of the developed models. For this purpose, it comprises a code generation facility [63]. As the Eclipse Modeling Tools solution is used in this thesis, the EMF is included anyway. Hence, a manual installation can be omitted. At startup, the services of EMF can be utilized immediately.

Building metamodels and models is not only restricted to the syntactical level, but also evokes the need of introducing constraints in terms of semantic aspects. Thus, the *OCL Tools* extension for Eclipse is installed in the development platform. Therefore, constraints like invariants or conditions can be added to the elements of the implemented metamodels and their models. The validation mechanism of EMF checks both the semantical and the syntactical correctness. In special editors, the metamodels are supplemented by the OCL fragments. The package is sufficient to cover the required range of OCL constructs.

In addition to the metamodeling and modeling, MDE approaches comprise M2M and M2T transformations. While the former are used to translate a PIM into a PSM, the latter define code generations based on PSMs. In this context, a set of Eclipse extensions can be integrated which implement these MDE principles. The M2M transformations are implemented with the package *QVT Operational* which conforms to the homonymous OMG standard. On the other hand, the generation of executable program code during the M2T transformation is done by means of

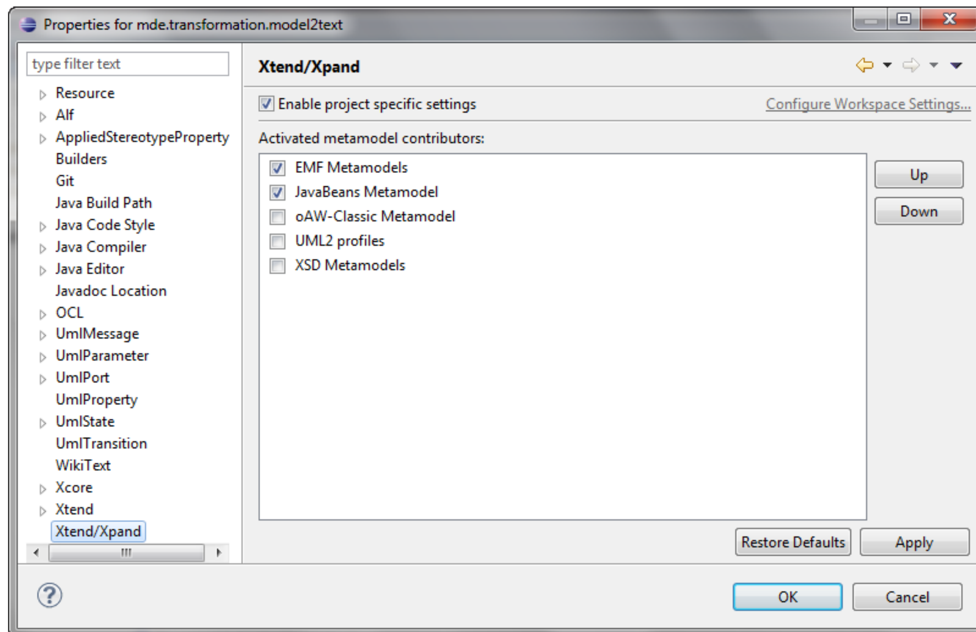

Figure 5.2: Xpand properties

another Eclipse plug-in. This is called *Xpand* which was initially developed by oAW and is now part of the Eclipse Modeling Project [18, 55]. Xpand supports template-based transformations and enables the generation of different kinds of output (e.g. source code, documentation, test cases).

After the installation of the plug-ins, including Java, EMF, OCL Tools and QVT Operational, most of these operate directly without modifying any configuration properties. Xpand, on the other hand, needs an adjustment of its settings. Figure 5.2 shows the Xpand properties window of Eclipse. As mentioned earlier, the oBIX metamodel defines the language for the source model of the code generation in this thesis and uses custom types for some class attributes which exist in separate Java classes. Without selecting *JavaBeans metamodel* besides *EMF metamodels* as activated metamodel contributor, the Xpand template is not able to address these particular attributes, and compiling errors will occur.

Eclipse organizes the individual user projects in so called *workspaces*. The actual list of projects and their assignment to different workspaces is subject of Section 5.2. However, it has to be noted already in advance that this implementation uses two workspaces since some projects provide their services (e.g. generated model editors) only at runtime to constitutive projects. Thus, the former and the latter are integrated in different workspaces considering their dependencies. Details are explained later on.

In summary, the presented extensions in combination with the Eclipse IDE allow the development of a model-driven approach according to the MDA initiative to model BASs and finally integrate them into an interfacing technology.

5.1.2 Gateway implementation

The gateway represents the interface between the BA network and remote applications that manage the network from the outside. Therefore, the BAS has to be integrated into such a gateway server. On the one hand, the implementation of the MDA workflow by the prepared development environment leads to a runnable program code which is able to produce an image of the BAS in the integration technology. On the other hand, a suitable implementation of a gateway must be selected to finally execute the generated code and offer services to manage the BA network behind the interface.

In this work, a research implementation of the oBIX standard is used which has been developed by the Automation Systems Group[1] at Vienna University of Technology. This integration middleware for the IoT is entitled *IoTSyS*. A gateway concept and a communication stack offer appropriate interfaces for embedded devices and smart objects like BASs. IoTSyS consists of a number of separate applications and projects which work together in an Open Services Gateway initiative (OSGi) environment. The following listing itemizes some of these subprojects [29]:

IoTSyS-Gateway is the oBIX server that provides WS endpoints for REST and SOAP. It uses HTTP and CoAP protocol for communication.

IoTSyS-oBIX is a modified version of the already introduced oBIX toolkit [50]. Amongst others, the implementations of the oBIX classes are located in this project.

IoTSyS-Common accomodates common interfaces and classes used in other IoTSyS projects.

IoTSyS-Calimero is a library wrapper for the Calimero framework [13, 42].

IoTSyS-KNX contains the implementations for connecting KNX networks with the gateway. For example, the network instantiations or the KNX datapoint implementations are part of this project. The proof of concept implementation of the underlying thesis is mainly concerned with this software bundle.

IoTSyS-BACnet4J is a library wrapper for BACnet4J similar to IoTSyS-Calimero.

IoTSyS-BACnet comprises all implementations for integrating networks of the Building Automation and Control Networks (BACnet) standard.

Figure 5.3 shows a UML package diagram containing the listed components and their dependencies. Due to all these projects, the gateway implementation offers the needed interface to realize the targeted approach. It provides interaction with both the BASs and the remote BMSs. In terms of a preferably simple interface, the REST paradigm offers a suitably small set of operations which is intended in the oBIX standard. Section 2.2.2.3 has already pointed out the key features and constraints of REST.

The HTTP interface of the oBIX gateway implementation provides three RESTful services for interacting with an integrated BAS. These operations are `GET`, `PUT` and `POST`. `GET` is used to retrieve information from the server. Such a call contains the requested URI to discover the

[1] `https://www.auto.tuwien.ac.at`

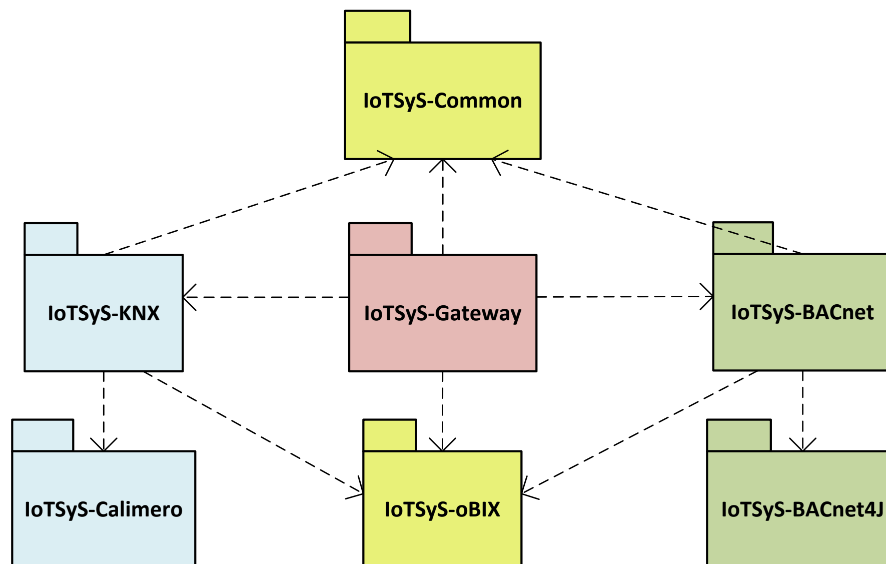

Figure 5.3: IoTSyS packages

information within the server objects. The result depends on the addressed object. According to the convention, the `GET` method should not take any action besides retrieval of information. Together with `POST` and `PUT`, `GET` is an idempotent method. The side-effects are always the same, irrespective of the number of identical requests. `PUT` is used to store the enclosed entity under the given URI. In contrast, the enclosed entity of a `POST` method request is defined as a new subordinate of the addressed resource [24]. By means of these simple methods, the gateway offers a basic yet powerful interface. The developers of BMSs do not need to pay attention to special protocols or complex function calls.

The gateway implementation is supported by the oBIX toolkit library. This package is available as Java implementation which is another reason for the focus on Java in the development environment. Although the oBIX standard can be developed for any programming language or any platform, an available and approved implementation eliminates the necessary effort for a new development.

As the proof of concept implementation deals with a KNX network as BAS, this paragraph takes a closer look at the KNX specific projects of the IoTSyS platform. One project (*IoTSyS-Calimero*) contains the Calimero library for Java to communicate with KNX devices. It is possible to read and write messages using the information exchange protocol of KNX. An interface on a higher abstraction level is provided to conceal the complex implementation details. The second KNX project (*IoTSyS-KNX*) uses this library and supports the integration of KNX components or even entire networks into the oBIX gateway. Implementations of KNX specific datapoints are part of this project. For instance, the binary datapoint *DPST-1-1* can be found in the set of classes. The member functions realize the communication with the actual datapoint and prepare data for further processing inside the gateway. These classes are derived from more common interfaces and super classes. Listing 5.1 shows the `writeObject` method of the mentioned

datapoint implementation. Here, line 11 indicates the Calimero function call to write a message on the KNX bus.

```java
public void writeObject(Obj obj)
{
  if (this.value().isWritable())
  {
    super.writeObject(obj);

    this.value().setNull(false);
    this.encoding().setNull(false);

    // now write this.value to the KNX bus
    connector.write(groupAddress, this.value().get());
  }
}
```

Listing 5.1: KNX DPST-1-1 implementation snippet

Besides the datapoint implementations, *IoTSyS-KNX* contains the so called *device loaders* for KNX. These classes are used to integrate KNX devices within the oBIX server. Which one of the various device loaders is executed, is defined via configuration files in the gateway. Public methods, which are inherited from a common device loader interface, initialize the desired components by instantiating appropriate oBIX objects. Furthermore, the connection to the BA network is established in this context. However, these device loaders are not restricted to KNX as other technologies like BACnet also use the device loader interface for integration. The code generation of this model-driven approach produces such a device loader class which can be immediately linked with the gateway.

In summary, the presented Java implementation of an oBIX gateway offers all necessary services for the proof of concept implementation. The additional libraries (e.g. Calimero, oBIX toolkit) are also available in Java. Thus, the overall development platform can be exclusively based on Java. The next section shows the implementation of the model-driven approach on the basis of the determined environment. Afterwards, details about the evaluation of this implementation are discussed in Chapter 6.

5.2 Realization

The overall, model-driven approach is separated into a set of single projects that are presented in this section. As already mentioned, the selected development environment, the Eclipse IDE, uses workspaces to include user-defined projects. In this approach, two different workspaces are created:

- First, one workspace contains those projects (e.g. libraries) which have no dependencies to other developed projects. If a project does not need the registration of other components during runtime, this project will be included in this workspace, too. Henceforward, the workspace in question will be called *development workspace*.

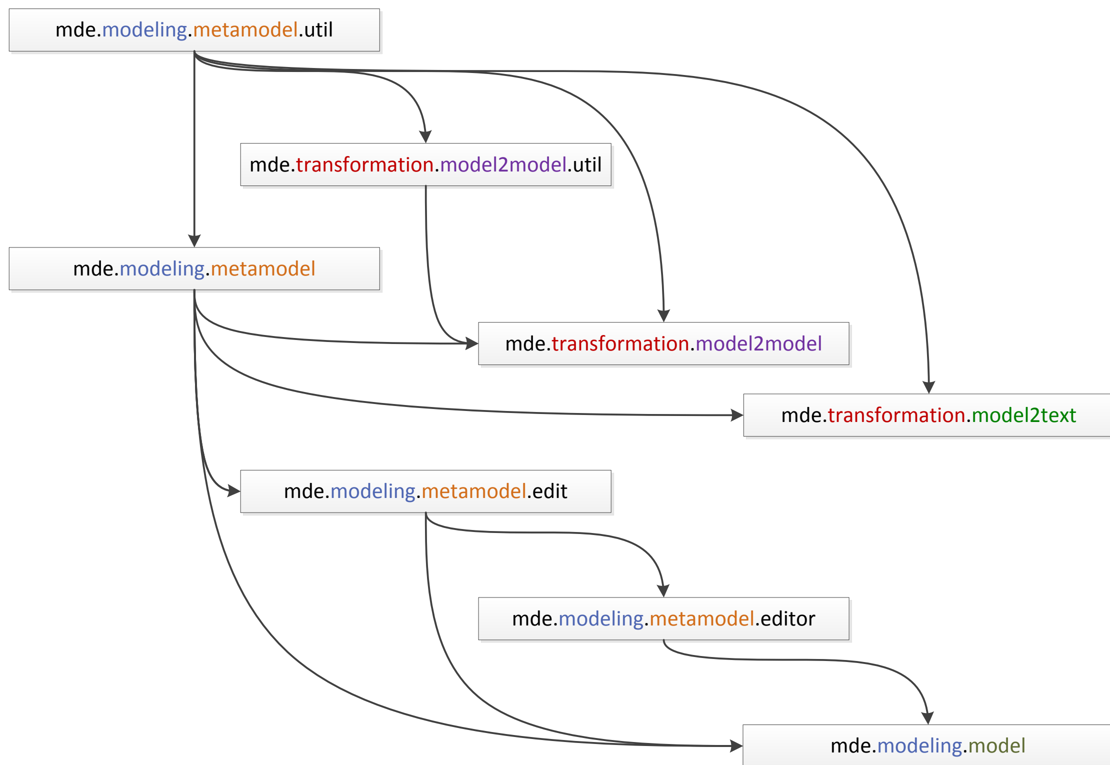

Figure 5.4: Workspace projects

- Second, all projects with runtime dependencies to earlier developed components are integrated into another workflow. For instance, the models need the registration of the superordinated metamodels to enable their creation and validation which is done by executing the metamodels as *Eclipse application*. From now on, this workspace will be called *runtime workspace*.

Before the individual projects are discussed in detail, they are summed up in the following listing for a better overview. For this purpose, Figure 5.4 supports the itemization by showing the dependencies between the different components. An arrow symbolizes a dependency relationship in which the project next to the arrowhead is dependent on the other project. Although the dependencies are sometimes transitive, the figure explicitly specifies all direct and indirect connections between the implemented components. The naming of the projects is similar to namespaces in order to simplify the project categorization. The common prefix *mde* stands for the pooling of all components in one single MDE or rather MDA implementation. Furthermore, there exists a branch for the modeling projects (*modeling*). On the other hand, there is also a branch for the transformations (*transformation*). Finally, the name concludes with the unique project name and an optional suffix *util* for the libraries. The membership to one of the two workspaces is also indicated in the subsequent list:

mde.modeling.metamodel contains the metamodels for BASs (modeling language definition for PIMs) and oBIX (modeling language definition for PSMs). The EMF and OCL Tools packages support the development process of these metamodels. Additionally, this project uses the classes of the metamodel library (*mde.modeling.metamodel.util*). It exports the generated metamodel specific source code for subordinated projects. The component is part of the *development workspace*.

mde.modeling.metamodel.edit is generated into the *development workspace* in order to provide a model editor in combination with *mde.modeling.metamodel.editor*.

mde.modeling.metamodel.editor consists of the remaining editor classes. These are also generated by *mde.modeling.metamodel* and are part of the *development workspace*. In both editor projects, no manual changes have to be made. Therefore, they are mostly omitted in the subsequent discussion of the realization details.

mde.modeling.metamodel.util is the library for the metamodel project and also for the transformation projects. It consists of a set of Java classes representing the custom types of the oBIX metamodel. Similar to the other metamodel projects, this component is located in the *development workspace*.

mde.modeling.model is based on the metamodel project and the generated editors. Thus, this component is found in the *runtime workspace*. It is a plain Eclipse project without any special features. PIMs and PSMs are the only elements within this project.

mde.transformation.model2model is the second project of the *runtime workspace*. As the name already suggests, it contains the model transformation from the PIM to the PSM. The project is dependent on the metamodel project, the metamodel library project and the model-to-model library (*mde.transformation.model2model.util*).

mde.transformation.model2model.util consists of a utility library with some special functions to support the model transformation, e.g. instantiation of custom oBIX types. The *development workspace* houses this project. The only dependency is the metamodel library with the actual implementation of the custom types.

mde.transformation.model2text is the last project in the MDA workflow. It comprises the template for the code generation. The oBIX metamodel definition and the custom types of the metamodel library are imported, yet this project is not integrated in the *runtime workspace*. Instead, it is part of the *development workspace* as only the metamodel file and no registration within the Eclipse environment is needed.

The next subsections are concerned with the detailed implementation of these projects respectively the four main components of the model-driven approach, i.e. metamodels, models, model transformation and code generation. The focus lies on the constructed files and their integration into the development environment.

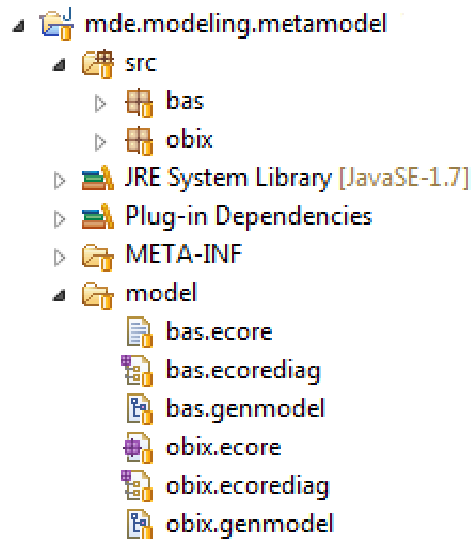

Figure 5.5: Metamodel project structure

5.2.1 Metamodels

The metamodels form the basis for further development of MDA transformations. The project *mde.modeling.metamodel* combines both the BAS metamodel and the oBIX metamodel. In Eclipse, an empty EMF project has been created. Within this project, two Ecore diagrams have been attached. Figure 5.5 shows the structure of the Eclipse project with all relevant folders and files. The *.ecore* files contain the serialized metamodel information while the *.ecorediag* file is used to store the graphical appearance. Both are generated by applying an Ecore diagram.

In general, unimportant files (e.g. *build.properties*, *plugin.properties* or *plugin.xml*) have been removed from the subsequent project structure figures in order to increase clarity.

There are two ways of building EMF metamodels. On the one hand, an Ecore model can be edited directly with a common text editor or a generic model editor. On the other hand, an Ecore diagram enables the possibility to design the model in a graphical way. The changes are transferred to the Ecore model. A tool palette in this graphical editor contains all the Ecore elements which have already been mentioned in Section 3.2.2. Although the chosen modeling method is irrelevant, the graphical editor gives a better overview of the model during the development process. In Figure 5.6, this editor can be seen. Elements are created by drag and drop from the palette to the worksheet. Properties are edited via the corresponding view which is located in the lower part of the figure. The actual elements of the metamodels have already been discussed in detail in Chapter 3. Thus, the focus of this section is only on using the development environment to create the desired files.

The semantic constraints in the form of OCL invariants are created with a special editor that is provided by the OCL Tools extension. It is called *OCLinEcore Editor* and displays the content of the Ecore file in OCL syntax. At first, the constraints are inserted in the correct position. While saving the file, the OCL constructs are converted to Ecore annotations. Therefore, the constraints

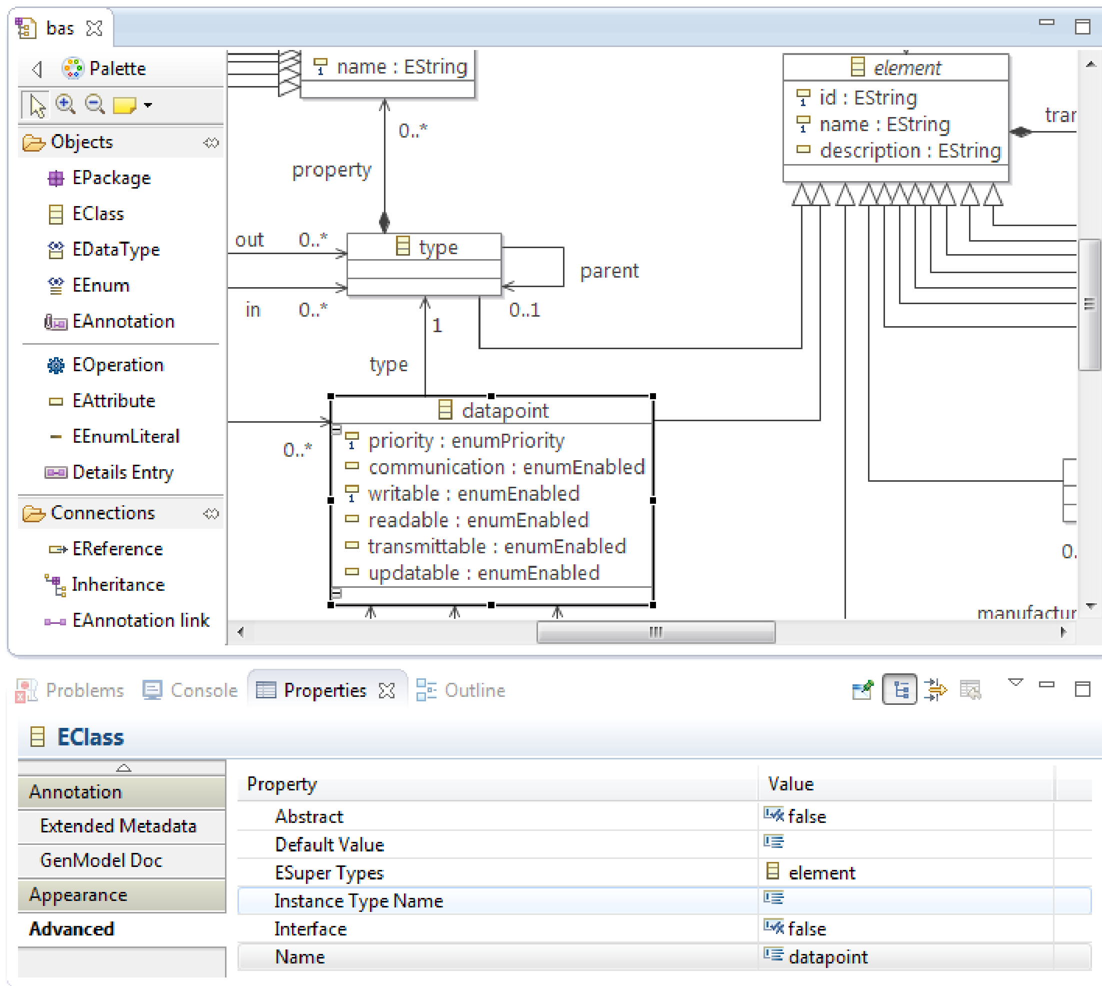

Figure 5.6: Ecore diagram editor

can also be edited in the graphical editor or the XML serialization of the Ecore model.

Once the metamodels are completed, Java classes can be generated to provide editors for these metamodels. For that purpose, *EMF generator models* are created by the Eclipse file wizard. The files with the suffix *.genmodel* belong to an Ecore model, and enable for the generation of model code, edit code, editor code and test code. Model code is created in the *src* folder of the metamodel project itself. The package is named in accordance to the Ecore model settings by default. Hence, the packages in this approach are called *bas* and *obix*. Each element in the metamodel becomes an interface. Additionally, implementation classes for the interfaces and some other classes are generated. The option to generate edit code spawns a new project *mde.modeling.metamodel.edit*. Likewise, the editor code is created in *mde.modeling.metamodel.editor*. The source code packages of these projects are automatically added to the exported packages in the runtime configurations of their manifest files. These

three components are needed to create models conforming to the metamodels. It has to be noted that the metamodel project and the two editor projects are Eclipse plug-ins. To offer their services, an Eclipse application is instantiated which launches all the necessary plug-ins. Then, the new workspace (*runtime workspace*) can use the (generic) editors to build BAS and oBIX models (see Section 5.2.2). If there are changes in the metamodels, the generator model can be easily reloaded.

```
public class DateTime
{
  private java.util.Date date;

  public DateTime(String value) throws Exception
  {
    ...

    SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd'T'hh:mm:ss.SSSz");
    date = f.parse(value);

    ...
  }

  @Override
  public String toString()
  {
    ...

    return new SimpleDateFormat("yyyy-MM-dd'T'hh:mm:ss.SSSz").format(date);
  }
}
```

Listing 5.2: oBIX DateTime type snippet

Due to the fact that the oBIX metamodel uses custom data types, implementations of these types are necessary. A separate project has been created to encapsulate these class files in terms of a more modular composition. Thus, the Java project *mde.modeling.metamodel.util* consists of six Java classes aggregated in the package *obix.type* which is exported in the manifest file. The structure of the Eclipse project is shown in Figure 5.7. The implementations have a similar construction. First, there exists only one constructor with a character string as input. This string represents the text inserted in the corresponding field of the IDE's properties view. Second, the method `toString` is overridden to return the internal value of the object. Hence, this value can be displayed in the properties view again. Exemplarily, Listing 5.2 shows the oBIX type `DateTime`. Here, the input value is parsed to a variable of the Java type `Date` regarding the correct date format. The `toString` method returns a formatted string of the stored date value. As the other types differ only in details, they are not listed in this section, but they are available online [45].

The library is included in the dependencies list of the metamodel project. The expressiveness of Java can be used to process the input and output values since the custom types are realized as

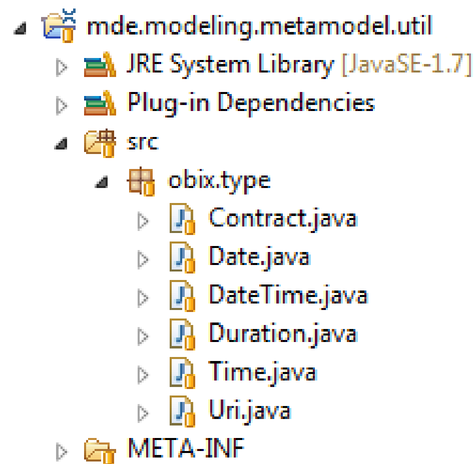

Figure 5.7: Metamodel library project structure

common classes. For instance, the class `Contract` represents the list of contracts in an oBIX object. But this textual sequence is split and stored in a Java collection for further processing. A method for checking the correctness of inserted URIs may be a possible extension of the type `Uri`, but such additional functionality is not focused in this thesis.

5.2.2 Models

After execution of the Eclipse application to enable the usage of metamodel editors, the general Eclipse project *mde.modeling.model* has been created in the *runtime workspace*. It contains both BAS models and oBIX models. Figure 5.8 shows the folder structure and the integrated network models of the Eclipse project.

While the BAS models are created with one of the two approaches for network modeling (see Section 4.2), the oBIX models are the result of MDA model transformations. In addition to the BA network models conforming to the BAS metamodel, the project comprises two model libraries for this metamodel. Both libraries are used by the BA networks as they provide necessary meta information:

- A general library consists of a set of enumerations and units. In principle, these models can be used in all types of BA network models because they are independent of any technology specific issues. For instance, the unit *celsius* is part of this library.

- A library for KNX has been developed in addition. Basically, libraries for any technology or any vendor will be possible, if a corresponding network is modeled with the BAS modeling language. In this thesis, a KNX network is part of the proof of concept implementation. Therefore, the necessary KNX datapoint types, parameters and encodings have been created in this library.

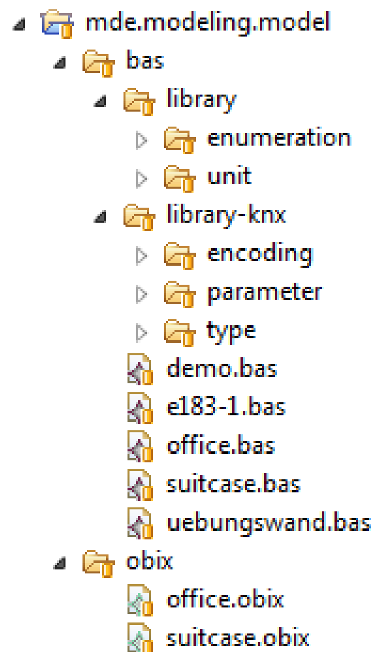

Figure 5.8: Model project structure

The *EMF Model Creation Wizard* for BAS models or oBIX models can be used to create new instances of the available metamodel elements. First, the root object of the model has to be chosen. For instance, a unit should be modeled as PIM. Then, the `unit` object of the BAS metamodel becomes the root model object. Next, the model can be designed according to the requirements as long as it is within the scope of the modeling language. The generic model editor is already limited to the permitted actions (e.g. the set of allowed child nodes or the range of an attribute value).

Models can also be created and modified with individual editors or standard text editors. In such cases, a subsequent validation with respect to syntactical and semantical correctness is necessary. For this reason, the model editors offer a method to check the accordance with the underlying metamodel. During this validation process, all references to models in other files (e.g. units, enumerations) are resolved, and the external models are inspected, as well. Errors in linked models lead to an error in the entire validation process. The syntactical compliance refers to the accurate usage of the modeling language concepts which are defined in the metamodels. In addition, OCL constraints are tested in the semantical part of the examination. In case of an error, the editor gives notice of the reason respectively the location of the problem.

5.2.3 Model transformation

This section is concerned with the realization of the M2M transformation from a PIM to a PSM. As stated above, the implementation of the model transformation is split in two distinct projects. On the one hand, there is the actual transformation project *mde.transformation.model2model*

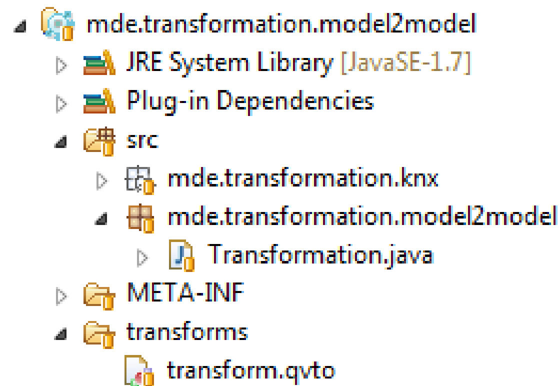

Figure 5.9: Model transformation project structure

containing the transformation file with the mapping rules. On the other hand, the transformation needs a special library with supporting functions which is located in the utility project *mde.transformation.model2model.util*.

Figure 5.9 shows the structure of the model transformation project. It has been created as an *Operational QVT Project* in the thesis' *runtime workspace*. The transformation rules are contained in the file *transform.qvto* which represents an *Operational QVT Transformation*. A separate Java class *Transformation* has been developed to enable the execution of this transformation without any additional Eclipse *run configuration*. Thus, a standalone execution of the model transformation is feasible. As known from the realization overview, the project is also dependent on the metamodels and the metamodel library with the implementation of the custom oBIX types. Moreover, the package *mde.transformation.knx* consists of an XSLT stylesheet to transform the KNX network data from the ETS4 to a PIM conforming to the BAS metamodel. This is discussed in more detail in Chapter 6.

```
1  import m2m.qvt.oml.ExampleJavaLib;
2
3  modeltype Bas uses 'http://auto.tuwien.ac.at/bas';
4  modeltype Obix uses 'http://auto.tuwien.ac.at/obix';
5
6  transformation transform(in bas : Bas, out Obix);
7
8  main()
9  {
10   bas.rootObjects()[bas::network].map network2obix();
11 }
```

Listing 5.3: QVT header

All transformation rules for converting a BAS model into an oBIX model are grouped in mapping definitions (mappings). The *.qvto* file has to import the source metamodel and the target metamodel. In line 1 of Listing 5.3, the import statement for the transformation library is

shown before the metamodel types are defined. The transformation is called *transform*, and it has one input parameter and one output parameter as can be seen in line 6. The function `main` is the root entry point for the transformation. There, the mapping for all the source model's root objects of type *network* is started. Admittedly, there are a lot of possible root objects (e.g. unit, type, enumeration) not only due to the BAS model libraries. However, the transformation of this thesis covers only the translation of BA networks to demonstrate the model-driven workflow for the integration of BASs into a gateway implementation.

```
1  mapping bas::network::network2obix() : Obj
2  {
3    name := "objects";
4
5    var currentNetwork : Obj := object Obj
6    {
7      name := self.id;
8      display := self.description;
9      displayName := self.name;
10     is := createContract("bas:Network");
11     uri := createUri("/networks/" + getUri(self.name));
12   };
13
14   var viewDatapoints := self.datapoints.map datapoints2obix(currentNetwork.
       uri);
15   var viewEntities := self.entities.map entities2obix(currentNetwork.uri);
16   var viewFunctional := self.functional.map functional2obix(currentNetwork.
       uri);
17   var viewTopology := self.topology.map topology2obix(currentNetwork.uri);
18   var viewBuilding := self.building.map building2obix(currentNetwork.uri);
19   var viewDomains := self.domains.map domains2obix(currentNetwork.uri);
20
21   var translations : _List := object _List
22   {
23     name := "translations";
24     uri := createUri("/translations");
25     of := createContract("obix:List");
26   };
27
28   result.Obj := result.Obj->union(self.datapoints.datapoint.map
       datapoint2obix(viewDatapoints.uri)->asSet());
29   result.Obj := result.Obj->union(self.entities.entity.map
       entity2obix(viewEntities.uri, viewDatapoints.uri)->asSet());
30   result.Obj := result.Obj->insertAt(1, viewDomains);
31   result.Obj := result.Obj->insertAt(1, viewBuilding);
32   result.Obj := result.Obj->insertAt(1, viewTopology);
33   result.Obj := result.Obj->insertAt(1, viewFunctional);
34   result.Obj := result.Obj->insertAt(1, viewEntities);
35   result.Obj := result.Obj->insertAt(1, viewDatapoints);
36   result.Obj := result.Obj->insertAt(1, translations);
37   result.Obj := result.Obj->insertAt(1, currentNetwork);
38 }
```

Listing 5.4: QVT network mapping

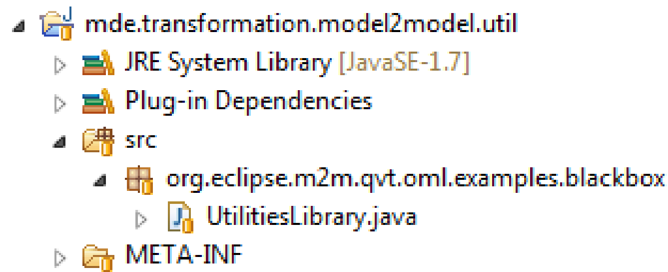

Figure 5.10: Model transformation library project structure

The target oBIX model does not have a single root node, but consists of several autonomous elements. For instance, while the network object solely contains reference objects for its views, the actual view objects exist independently of this parental network object. Thus, an artificial root element has to be generated to enable the integration of all independent objects (e.g. views, datapoints, entities, translations) in one single oBIX model. This is done in the mapping of the network element which is shown in Listing 5.4. The mapping takes a BAS network object and returns an oBIX `Obj`. This return value becomes the artificial root node with the simple name *objects*. Subsequently, all directly subordinated elements are constituted as variables either by calling other mapping definitions (e.g. views) or by creating a completely new object (e.g. network, translations). The listing shows these statements in the lines 5 to 26. The keyword *self* provides access to the input element of the current mapping while *result* addresses the return value.

All generated objects are added to the list of child objects of the result by inserting them in the collection `result.Obj`. Furthermore, the datapoints and entities are also added to this catalog which is listed in lines 28 and 29. The result is a model with the root node *objects* and a large set of child elements containing the network object itself, the views for building, topology, domains and functional, the individual datapoints and entities and three list objects consisting of datapoints, entities and translations. As the transformation rules have been discussed earlier in Chapter 4, they are not explained in this section. The other mappings in this transformation file implement the theoretical and abstract rules from Section 4.3. While this clipping omits lots of mappings and statements, the entire QVT file can be found online [45].

The Java class for a standalone execution of the model transformation (*Transformation*) offers two more advantages besides the execution without any Eclipse configuration. At first, the link to the upstream library can be managed with a small set of Java code lines. Thus, it is easily extensible for other libraries. In addition, the metamodels, the input model and the output model can be specified in just a few statements. Second, the adding of the XML Schema location in the header of the oBIX output model can be enabled by this form of transformation call. This is necessary for the subsequent code generation where a reference to the oBIX metamodel in the template's source model is needed for processing the meta code. Therefore, a standalone model transformation is obviously the best choice for this proof of concept implementation.

Next, the library for the transformation is examined. Again, Figure 5.10 shows the project structure with the library file *UtilitiesLibrary.java*. The project has been built as *Black-box*

Library Definition which is an Operational QVT example offered by the Eclipse QVT package. Although it is possible to create the library as an empty project, the example project already includes all necessary plug-in configurations and dependencies. No additional configuration is required. The predefined custom oBIX types form a prerequisite of the library project. The utility class itself is exported as extension in the plug-in settings of the project. For the sake of simplicity, the default settings of the extension point as well as the package denomination were kept. The utility class can be divided in a few main parts:

- The management of URIs is realized by means of a few methods. If two or more elements (e.g. datapoints) get the same transformed oBIX URI, these URIs are consecutively numbered to provide uniqueness. The library stores the generated URIs in a hashmap and returns a unique string to the calling transformation. Moreover, the passed back URI is free of special characters like blanks or German mutated vowels.
- Two methods simulate an incremental counter to enumerate elements during the transformation process. One function resets the counter, and the other one increments the returned value with each call.
- For each custom type of the metamodel library (e.g. DateTime), a method which creates and initializes an object of this type exists. Finally, the result is returned to the transformation.
- A logging method enables the output of messages in the console window of the Eclipse IDE during the execution of the standalone transformation.

```
1 public Contract createContract(Contract contract, String value)
2 {
3   if (contract == null)
4     contract = new Contract();
5
6   contract.addContract(value);
7
8   return contract;
9 }
```

Listing 5.5: Model transformation utility function

The extension is loaded and linked within the standalone transformation class before the actual transformation is invoked. Hence, the model transformation rules have access to the various library methods. Consequently, this enables more powerful solutions for the MDA workflow step. Only the signature of the utility functions is known within the QVT context. This is the reason, why it is called a black box library. For instance, the function `createContract` is called in line 10 of Listing 5.4. The resulting object is assigned to the `is` attribute of the network object. In the end, Listing 5.5 shows the implementation of this `createContract` method. The `Contract` object is only created, if the input parameter is empty. Otherwise, the existing contract is expanded by the second input parameter to form a list of contracts. The entire library file is listed online [45].

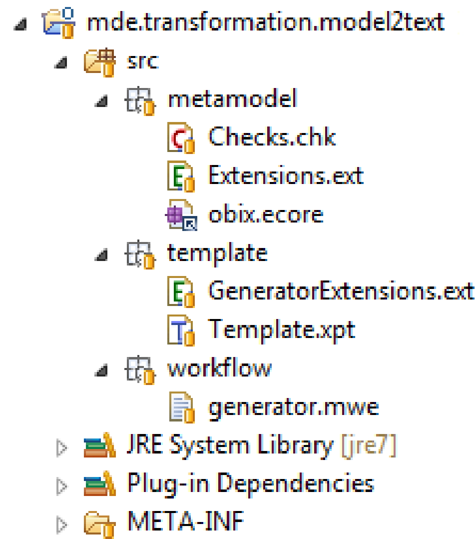

Figure 5.11: Code generation project structure

5.2.4 Code generation

The last building block of this model-driven approach is the generation of executable program code. In MDA, this step is called M2T transformation. Xpand is used as underlying technology to realize this transformation. The implementation is located in the Eclipse project *mde.transformation.model2text* which has been created as default Xpand project with the Eclipse project wizard. Additionally, the option *Generate a sample EMF based Xpand project* has been activated. Thus, the created project is sufficiently configured for the needs of the implemented code generator. Xpand is part of the family of languages launched by the former oAW [55]. The other languages (*Xtend*, *Check*) can also be embedded into this Xpand project.

Within the source folder of the project, the generated files are partitioned into three subfolders. The folder *metamodel* contains a *Check* file and an *Xtend* extension file by default. As these languages are not used within this thesis, the files have been emptied. The third file in this folder is a link to the oBIX metamodel. Therefore, the metamodel can be imported in the template file which is located in the *template* folder besides another empty *Xtend* file. Finally, the subfolder *workflow* consists of a Modeling Workflow Engine (MWE) script for the execution of the code generation. There the source model and the location of the output are specified. Moreover, the metamodel is loaded and the generation is started in the script. In addition, a prior check of the model correctness is arranged. If the *Check* file is not empty, the specified constraints in this file will be used to inspect the source model. The final project structure is illustrated in Figure 5.11.

As already pointed out, the aim of the M2T transformation is the generation of a Java class which contains all statements to integrate a BA network into the oBIX gateway implementation IoTSyS. This class is encapsuled in a homonymous Java file that is stored in the connector's namespace folder of the *IoTSyS-KNX* project by the MWE script. On top of the transformation template, the oBIX metamodel is imported. Next, the standard library for unique identifiers is

linked with the template to enable the creation of unique Java names for the instantiated oBIX objects.

```
«IMPORT obix»
«EXTENSION org::eclipse::xtend::util::stdlib::uid»

«DEFINE main FOR Obj»
«FILE "KNXDeviceLoaderETSImplGenerated.java" TO_SRC»

package at.ac.tuwien.auto.iotsys.gateway.connectors.knx;

...

public class KNXDeviceLoaderETSImplGenerated implements DeviceLoader
{
  @Override
  public ArrayList<Connector> initDevices(ObjectBroker objectBroker)
  {
    KNXConnector knxConnector=new KNXConnector("192.168.1.102",3671,"auto");
    this.connect(knxConnector);
    this.initNetwork(knxConnector, objectBroker);

    ...
  }

  ...

  private void initNetwork(KNXConnector knxConnector, ObjectBroker
      objectBroker)
  {
    ...

    HashMap<String,Integer> groupAddresses = new HashMap<String,Integer>();
    initAddresses(groupAddresses);

    HashMap<String,ArrayList<ArrayList<String>>> translations = new HashMap<
        String,ArrayList<ArrayList<String>>>();
    initTranslations(translations);

    «EXPAND obj(null, null) FOREACH Obj»

    ...
  }
}

«ENDFILE»
«ENDDEFINE»
```

Listing 5.6: Xpand main template definition

Listing 5.6 starts with these two commands and continues with the main template definition (`main`) in line 4. This template definition is specified for any oBIX `Obj`. Thus, it is entered

when reaching the previously introduced root object *objects* while scanning the source model. First, the template definition `main` specifies the name of the output file. Subsequently, the target text with the Java package definition, various imports (omitted) and the actual Java class are listed. This class implements the interface `DeviceLoader` which offers three methods. The gateway implementation uses this interface for the integration of single devices or entire networks into the middleware. The most relevant method is `initDevices` in which the private method `initNetwork` is invoked. Prior to that, a connection to the KNX network is established. No matter if the connection is sucessfully initialized, `initNetwork` first scans the source model for all used group addresses. These addresses are stored in a hashmap to reuse them in the subsequent creation of datapoint objects. Likewise, the translations are filtered and stored. Afterwards, the template definition `obj` is invoked. Again, this definition is specified for the oBIX class `Obj` and all its derived classes. Thus, `obj` is called for every child object of the current root model element. The meta code is separated from the target text by the special characters « and ». Irrelevant target text and meta code as well as the comments, which would be marked by `REM` and `ENDREM`, are omitted in the listings of this section. The full code can be found online [45].

```
1  «DEFINE obj(String root, String add) FOR Obj»
2  «IF !this.uri.toString().contains("/translations")»
3    «LET "_" + createUID() AS name»
4
5      ...
6
7      «this.metaType.name.split(":").get(2)» «name» = new «this.metaType.name.
           split(":").get(2)»();
8
9      ...
10
11     «IF !this.is.toString().contains("bas:DPST") && !this.is.toString().
           contains("bas:DPT")»
12       «EXPAND obj(name, add) FOREACH Obj»
13     «ENDIF»
14
15   «ENDLET»
16 «ENDIF»
17 «ENDDEFINE»
```

Listing 5.7: Xpand template definition

The code for instantiation of the actual network with all views, datapoints and entities is generated in the template definition `obj`. Listing 5.7 is a cutout of this part of transformation. The condition in line 2 excludes the translation objects. They do not become separate objects, but the texts are included in other multilingual elements during transformation. The command `LET` determines a block where a created unique identifier can be addressed by the variable `name`. The value is generated by calling the function `createUID` provided by the imported standard library. Line 7 shows the declaration of an oBIX object where its type is set based on the interpretation of the given meta code. The new object gets the generated identifier as

its name. After the Java keyword `new` written in target text, the call of the default constructor is inserted. In the end, the template definition is recursively called, i.e. also the child objects are processed the same way. The datapoints represent an exception as their childs are already included in the instantiated datapoint implementations (see Section 4.4).

The library project is dependent on the metamodel library to gain access to those attributes with a custom oBIX type. All in all, the M2T transformation is designed to convert an oBIX model (source model) containing a BA network and its components into an executable Java class that can be easily integrated in the oBIX gateway implementation. This is the last step of the seamless and transparent integration workflow stated in Hypothesis 2. The next chapter uses this implementation to evaluate the realized, model-driven approach. Without taking too many details in advance, the evaluation is based on a small KNX network with a few devices and datapoints. This network runs through the transformation workflow, and finally, its oBIX representation is tested by some use cases and sample requests.

5.3 Deployment

After the implementation of the metamodels and the transformations, the model-driven process is ready for execution. First, the BA network is mapped to MDA models and passed through the workflow before the source code is generated. This sequence can be found in Subsection 5.3.1. Afterwards, this code is executed in the oBIX gateway which is described in Subsection 5.3.2.

5.3.1 Workflow sequence

Reconsidering the three steps of the transformation workflow (see Chapter 4), the sequence of workflow phases during runtime can be specified the following way. In each step, the focus lies predominantly on the execution of the previously introduced files and projects within the development environment. Figure 5.12 illustrates the deployment sequence as a supplement to the list.

- Before any transformation can be executed, the BA network has to be modeled as a PIM conforming to the BAS metamodel. The step is called *network modeling*. For this purpose, the BAS metamodel and its editor plug-in are executed as an Eclipse application. In the resulting environment (*runtime workspace*), the models can be created and designed in accordance with the requirements of the desired interface. The generated *BAS model editor* supports this workflow phase regarding design and validation. Previous chapters and sections have already pointed out the different methods of creating and modifying the network model. However, the provided model editor is the most comfortable way. Regardless of the selected network modeling approach, the result is a file containing a PIM which represents a real BAS. In addition, the necessary library models containing the meta information of the network are laid down in this step.

- Next, the modeled network has to be transformed from a PIM to a PSM. In MDA, this is done by means of the M2M transformation or *model transformation*. In this thesis, the QVT based model transformation is invoked by executing the main function of the Java

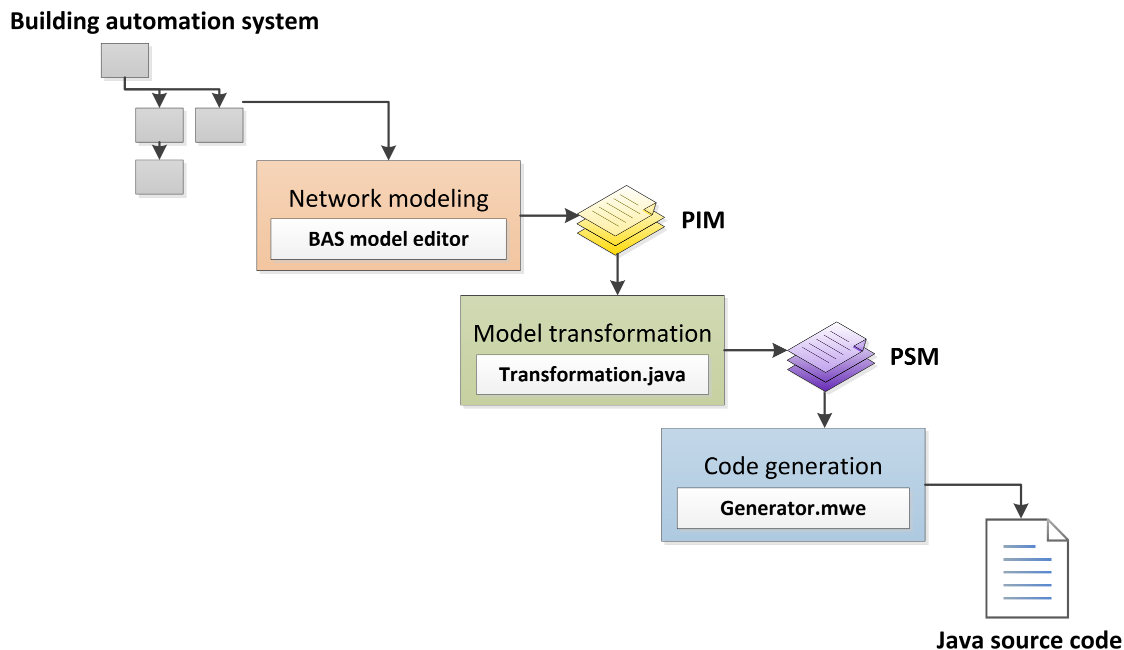

Figure 5.12: Deployment sequence

class *Transformation*. Before the conversion from the input PIM to the output PSM can take place, some static variables in the class have to be modified. First, both metamodels are set by the string variables `bas` and `obix`. Moreover, the distinct input model and the location of the output model have to be specified in the strings `input` and `output`. The invoked *transform.qvto* (`qvto`) also needs the plug-in extension of the M2M library (`plugin`). The current settings can be found in Listing 5.8. Finally, the class is executed as default *Java application*. If log messages or debug messages are contained within the transformation, they will be listed in the console view of Eclipse. Both the input model and the output model are located in the model project of the *runtime workspace*. The model transformation uses relative paths to address them.

```
1 public static final String bas="../mde.modeling.metamodel/model/bas.ecore";
2 public static final String obix="../mde.modeling.metamodel/model/obix.ecore";
3 public static final String input="../mde.modeling.model/bas/office.bas";
4 public static final String output="../mde.modeling.model/obix/office.obix";
5 public static final String qvto="transforms/transform.qvto";
6 public static final String plugin="../mde.transformation.model2model.util/
      plugin.xml";
```

Listing 5.8: Model transformation settings

- The last process step before the final program code is available, is the *code generation* or rather M2T transformation. Similar to the setting of the variables in the standalone model

transformation class, the necessary configuration in this phase is limited to the MWE script *generator.mwe*. The source model is specified in the *property* tag with the name *model*. The destination of the output is given in the *outlet* tag with the name *TO_SRC*. In this work, the path points to the KNX connectors folder of the *IoTSyS-KNX* project. Thus, the generated Java class can be directly executed in the gateway without any additional copying and linking. If the source metamodel changes as well, a link to the new one has to be established in the metamodel folder of the M2T project. The MWE script is run in Eclipse. No further configuration is needed as long as the metamodel is correctly imported in the template file. The runtime environment produces some logging information in the console view of Eclipse. Listing 5.9 shows the most important lines of this output.

```
1 WorkflowEngine      - running workflow: D:/mde.transformation.model2text/src/
      workflow/generator.mwe
2 StandaloneSetup     - Registering platform uri 'D:\'
3 CompositeComponent - Reader: Loading model from platform:/resource/mde.
      modeling.model/obix/office.obix
4 CompositeComponent - DirectoryCleaner: cleaning directory 'src-gen'
5 CompositeComponent - CheckComponent: slot model check file(s): metamodel::
      Checks
6 CompositeComponent - Generator: generating 'template::Template::main FOR
      model' => [TO_SRC:../iotsys-knx/src/at/ac/tuwien/auto/iotsys/gateway/
      connectors/knx, src-gen]
7 Generator           - Written 1 files to outlet TO_SRC(../iotsys-knx/src/at/ac
      /tuwien/auto/iotsys/gateway/connectors/knx)
8 WorkflowEngine      - workflow completed in 2236ms!
```

Listing 5.9: Console output of M2T transformation

5.3.2 Code execution

Afterwards, the generated source code as output of the MDA workflow is executed in the oBIX server. Thus, the configuration of the gateway has to be modified. As the generated class, which is derived from the device loader interface, has already been placed in the KNX project, only the *devices.xml* of the gateway project must be customized. It is sufficient to register the namespace of the new class as an additional device loader in the XML configuration. Next, the gateway can be run by executing the class *IoTSySGateway* of the gateway project (*IoTSyS-Gateway*). During the initialization process, the installed device loaders of the various integrated technologies are invoked. As a result, the oBIX objects listed in the generated file are instantiated and can be further managed via the HTTP or CoAP interface of the oBIX server. The way of accessing the objects, or reading and writing of KNX datapoints is subject of the case study in Chapter 6.

CHAPTER 6

Evaluation

The proof of concept implementation of the elaborated model-driven approach is evaluated as part of a case study. Thus, the hypotheses of Chapter 1 can be confirmed by means of an experimental KNX setup. It is demonstrated that the MDA approach can be applied to map BASs and finally integrate them into gateway technologies. Additionally, this chapter deals with related work in the field of interest and identifies open issues.

6.1 Case study

The aim of this evaluation is to demonstrate both the functional capability and the functionality of the developed model-driven approach. For this purpose, a case study is arranged. A sample KNX network is used as BAS, and the previously introduced integration middleware IoTSyS is established as gateway for accessing the BA network via a common interface. First, the general KNX engineering methodology is briefly considered. Afterwards, this common knowledge is applied to explain the experimental setup of the observed network. Before the integrated BAS is evaluated focusing on access scenarios via HTTP and CoAP, the mapping process from the available, technology specific data to the oBIX objects is discussed. The conversion of KNX engineering data into the model-driven approach is explained in general while some aspects are examined in detail.

The evaluated case study makes no claim to be complete, but aims to depict one possible application. Instead of KNX, any other BA technology can be used under the presumption that the technology's networks are able to be mapped to models that conform to the introduced BAS metamodel. In the end, the section mentions some use cases where such an MDE based procedure and a universal interface result in the reduction of development time and an increase of reusability during the implementation of subsequent BMSs. Hence, the advantages of such a standardized approach compared to an individual solution should be clear after reading this chapter.

6.1.1 Engineering of KNX

Based on the overview image stated in Chapter 1, the general components can be replaced by components of the actual proof of concept implementation. Figure 6.1 shows this revised illustration. The former generic BAS becomes a concrete KNX network engineered by the ETS4. The chosen automatic approach for network modeling uses an XSLT stylesheet which is based on the ETS4 export data. The network modeling step is examined in the following sections. Communication with the network is established by KNX frames that are exchanged between a KNXnet/IP router and the oBIX WS gateway. This integration server includes the KNX network representation. Further parts of this overview remain unchanged.

Regarding the KNX engineering process, the technology supports two possibilities for the configuration of devices and their bindings. While the Easy Mode (E-Mode) allows limited modifications of device parameters and links, complex building management functions can be implemented with the System Mode (S-Mode). In the latter, the ETS4 is used for planning and configuration of the network [39]. In the underlying case study, the use of the S-Mode is assumed. Subsequently listed points have to be taken into account while configuring a BA network based on the KNX technology. Following steps can be processed repeatedly:

1. Initially, the network has to be planned. The intention of the network and the needed physical devices are existent. Based on this, the devices are preliminarily integrated in the ETS4. Each KNX device has an order number for its identification in the manufacturer specific catalog. This number is used to locate the device as well as for its integration into the network. Imported devices contain all available datapoints and parameters. The device list of the ETS4 is the basis for further engineering.

2. Next, physical addresses have to be assigned to the devices. Prior to this, the KNX topology with its areas and lines is created. Afterwards, each device gets a tripartite address consisting of the area's address, the line's address and the individual device address. The address configuration is downloaded to the devices via the ETS4 programming function.

3. Then, the parameters of the devices must be set. For instance, the function of a push button or the threshold value of a CO_2 sensor can be determined in this phase of the process. The ETS4 lists the configurable adjustments per device in the device view. There, the settings are organized in various subgroups to offer a simplified view. Again, changes are downloaded to the devices.

4. Now, the network components are configured, but still no relationship between these elements exists. The various datapoints respectively function blocks of the devices have to be linked to build the desired distributed applications within the BA network. This is done by using the KNX grouping concept. Associated elements (e.g. datapoints for a push button and a light switching actuator) are combined in individual groups to enable the interworking of these components. In ETS4, datapoints are often called *communication objects*. Additionally, further views (e.g. building) are initialized in this step. However, these are not downloaded to the devices, but remain in the engineering software. They only provide additional information for monitoring purposes.

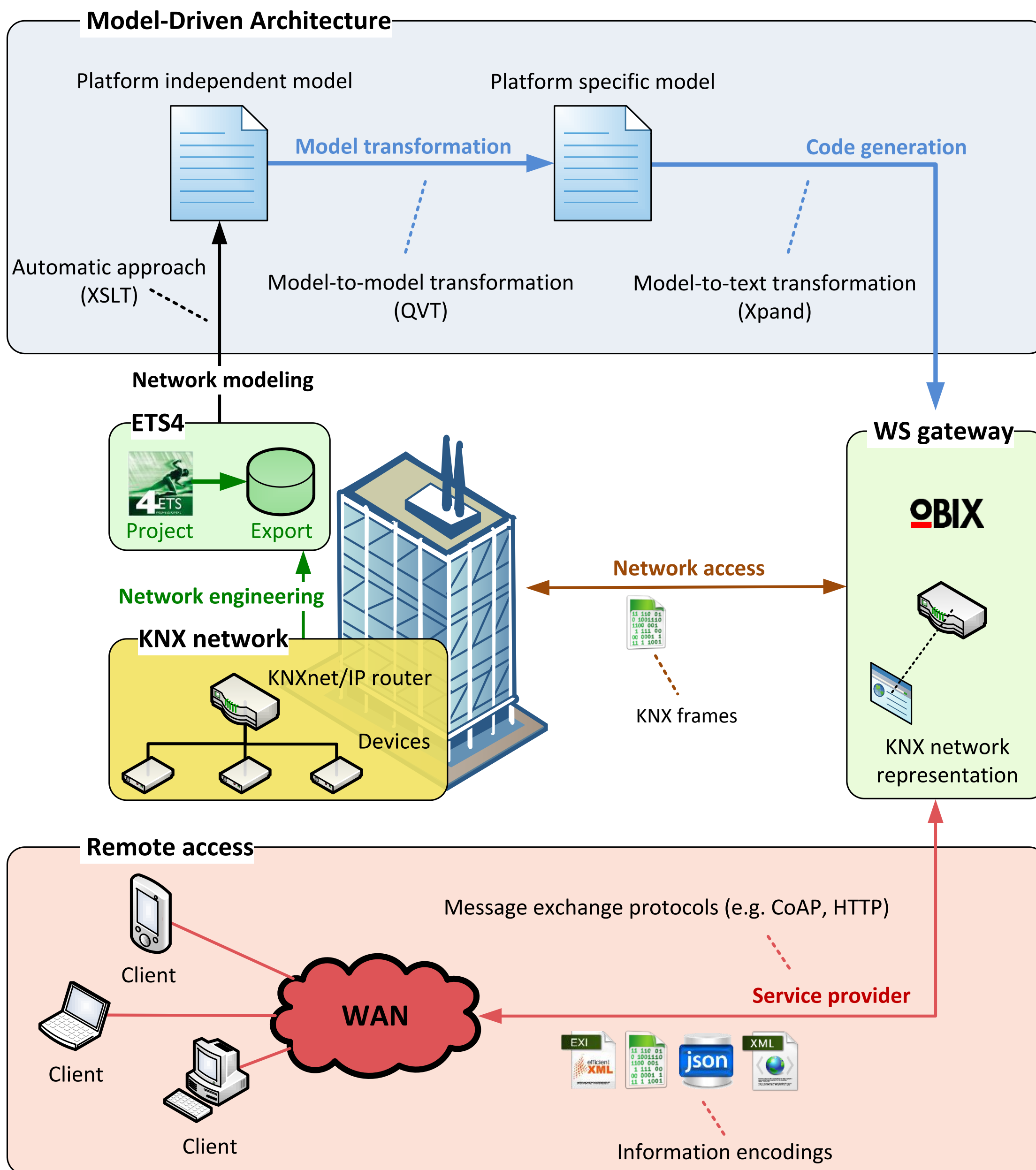

Figure 6.1: KNX specific overview

5. After downloading the configured application programs to the particular devices, the settings are taken over, and the network is ready for operation.

The ETS4, released by the KNX Association, is the platform for engineering KNX networks [19]. Different views in the user interface support the design of the network structure. Figure 6.2 shows a screenshot of the ETS4 user interface. In the upper half, the device view can

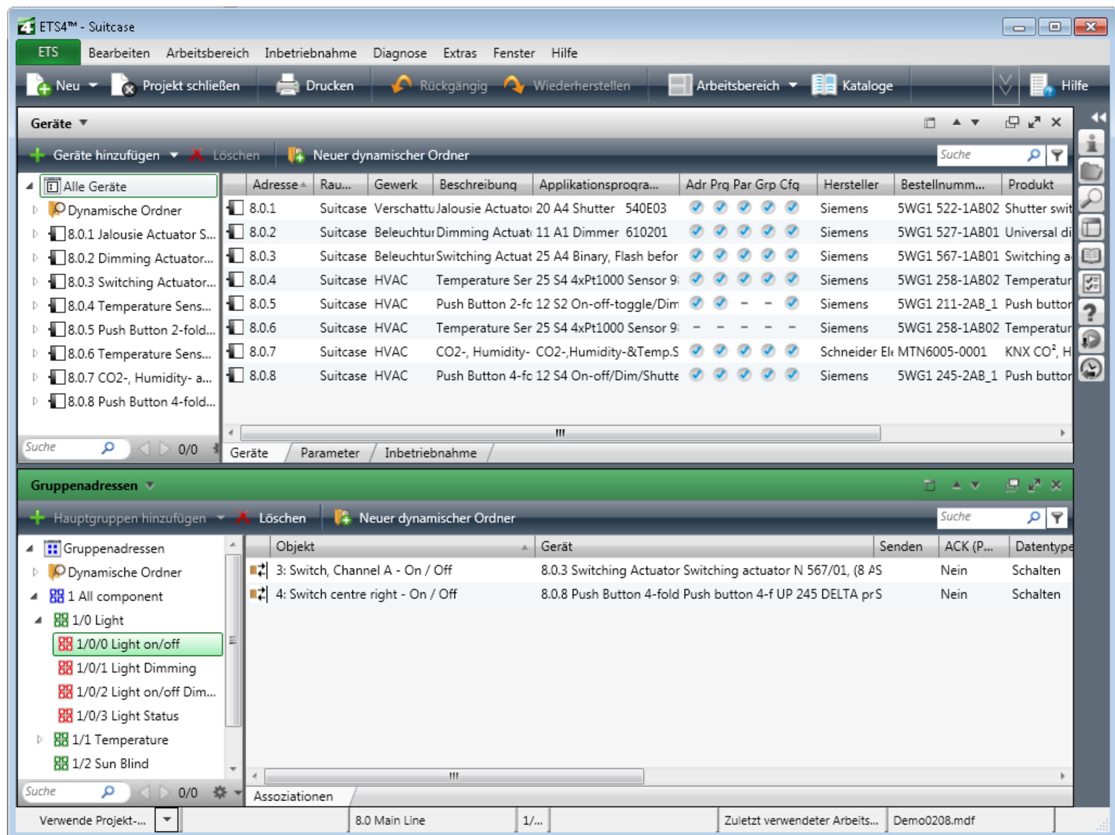

Figure 6.2: ETS4 user interface

be observed where devices and their datapoints can be managed and configured. In the lower half, group addresses can be initialized and modified. The catalogs of available devices that can be engineered are imported from manufacturer websites or other sources. In this thesis, the export function of ETS4 is utilized. Here, the entire project and its relevant meta information are packed in an archive file containing a set of XML files. This archive is the basis for further network modeling.

6.1.2 Experimental setup

The case study is based on a small KNX network. This experimental setup is embedded into a portable suitcase, and consists of some devices to test basic building automation scenarios. Figure 6.3 shows the utilized testbed. The BA network can be accessed via a Universal Serial Bus (USB) interface of Siemens (order number *5WG1 148-1AB11*) and a KNXnet/IP router of Weinzierl Engineering (order number *KNX IP Router 750*). Internally, the devices are connected via twisted pair (TP). Additionally, the suitcase possesses a direct power supply. The built in devices are listed in Table 6.1. While devices like power supply and IP router are omitted, only sensors and actuators are mentioned.

Address	Device	Manufacturer	Order number
8.0.1	Shutter switch	Siemens	5WG1 522-1AB02
8.0.2	Universal dimmer	Siemens	5WG1 527-1AB01
8.0.3	Switching actuator	Siemens	5WG1 567-1AB01
8.0.4	Temperature sensor	Siemens	5WG1 258-1AB02
8.0.5	Push button 2-fold	Siemens	5WG1 211-2AB_1
8.0.7	CO_2, humidity, temperature sensor	Schneider Electric	MTN6005-0001
8.0.8	Push button 4-fold	Siemens	5WG1 245-2AB_1

Table 6.1: KNX devices

The network is named *Office*, and it is engineered in an ETS4 project. The devices are imported by using integrated manufacturer catalogs. In these catalogs, each device can be found via its order number (see Table 6.1). The physical addresses are assigned in accordance to the second step of the general KNX engineering process (see Section 6.1.1). All devices are integrated in the main line of the network which is part of the area with address 8.

Next, the devices are divided into groups of domains. Thus, the shutter switch is linked with the domain *shading*. Both the dimming actuator and the switching actuator as well as the push buttons are integrated into the domain *lighting*. The temperature sensor and the combined CO_2, humidity and temperature sensor are part of the *HVAC* domain. For completeness, the building view is also initialized, but all devices are added to the same building part.

In this experimental setup, the main part of the network configuration is the instantiation of KNX groups. Datapoints of the devices are linked within these groups to introduce functional behavior. A threepart addressing scheme for the individual groups is chosen. One distinct main group with address 1 contains the five middle groups *Light* (1/0), *Temperature* (1/1), *Sun Blind* (1/2), *Other* (1/3) and *Buttons* (1/4). Below these middle groups, the actual groups are inserted. The following list explains the most important groups. Furthermore, these groups are shown in Figure 6.4. The image illustrates the network topology with the devices and their addresses. In addition, the groups are visualized by dashed lines between the devices. The groups for the CO_2 threshold values and the switches of the 4-fold push button are omitted in this list and in the figure.

1/0/0 *Light on/off* links channel A of the switching actuator with the center right button of the 4-fold push button. The function of the button is set to *On/Off*.

1/0/1 The group *Light dimming* establishes a connection between the brighter/darker dimming datapoint of the dimming actuator and the outer right button of the 4-fold push button. Here, the button's function is set to *Dimming, On/Off*.

1/0/2 *Light on/off dimmer* enables switching on and switching off of the dimming actuator. The same push button as in the previous group is used.

1/0/3 *Light status* gives information about the 8-bit value of the dimming actuator. Requests on this group address return the actual value of the dimming brightness.

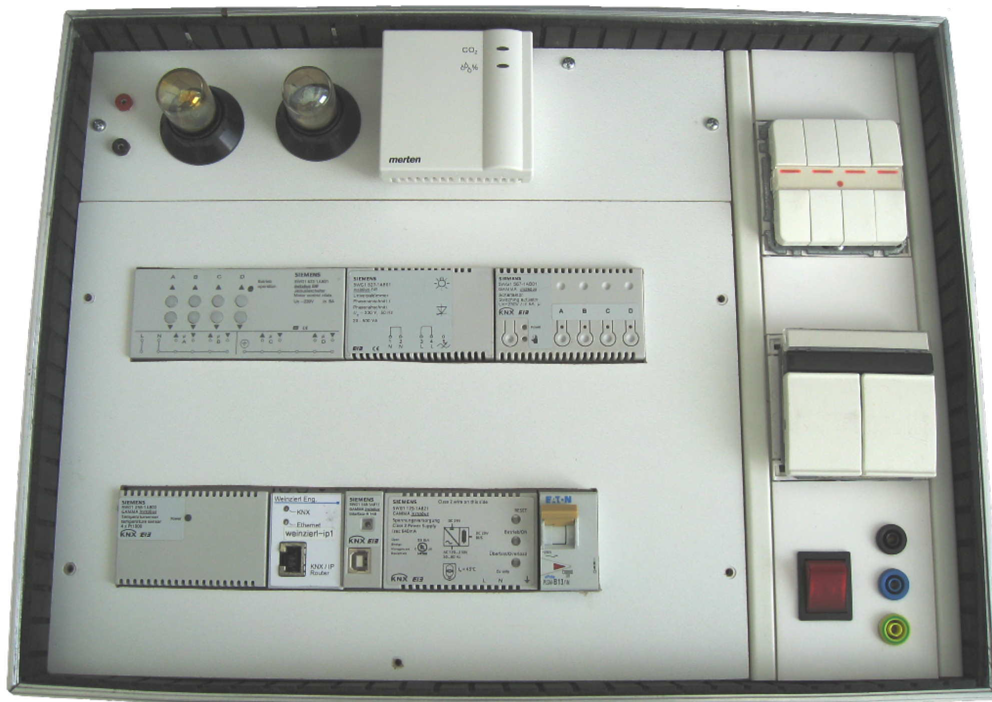

Figure 6.3: KNX testbed

1/1/0 – 1/1/3 The groups *Temperature 1–4* contain the temperature channels A to D from the temperature sensor. Each group includes one of these channels.

1/3/1 The CO_2 value of the corresponding sensor is provided in group *CO2*. No other datapoints are part of this group.

1/3/2 Likewise, *Relative humidity* includes the sensor's datapoint for the relative humidity value.

Besides the KNX network, the oBIX gateway has to be configured. The file *devices.xml* in the *IoTSyS-Gateway* project has to be modified. A new node of type *device-loader* is needed in this XML settings to enable the execution of the generated program code during server startup. Listing 6.1 states this additional configuration entry. As already mentioned, the gateway is executed as Java application on a local computer. At runtime, the WS interface is accessible via the HTTP port 8080 or the CoAP port 5683.

```
1 <device-loader>at.ac.tuwien.auto.iotsys.gateway.connectors.knx.
      KNXDeviceLoaderETSImplGenerated</device-loader>
```

Listing 6.1: oBIX gateway configuration

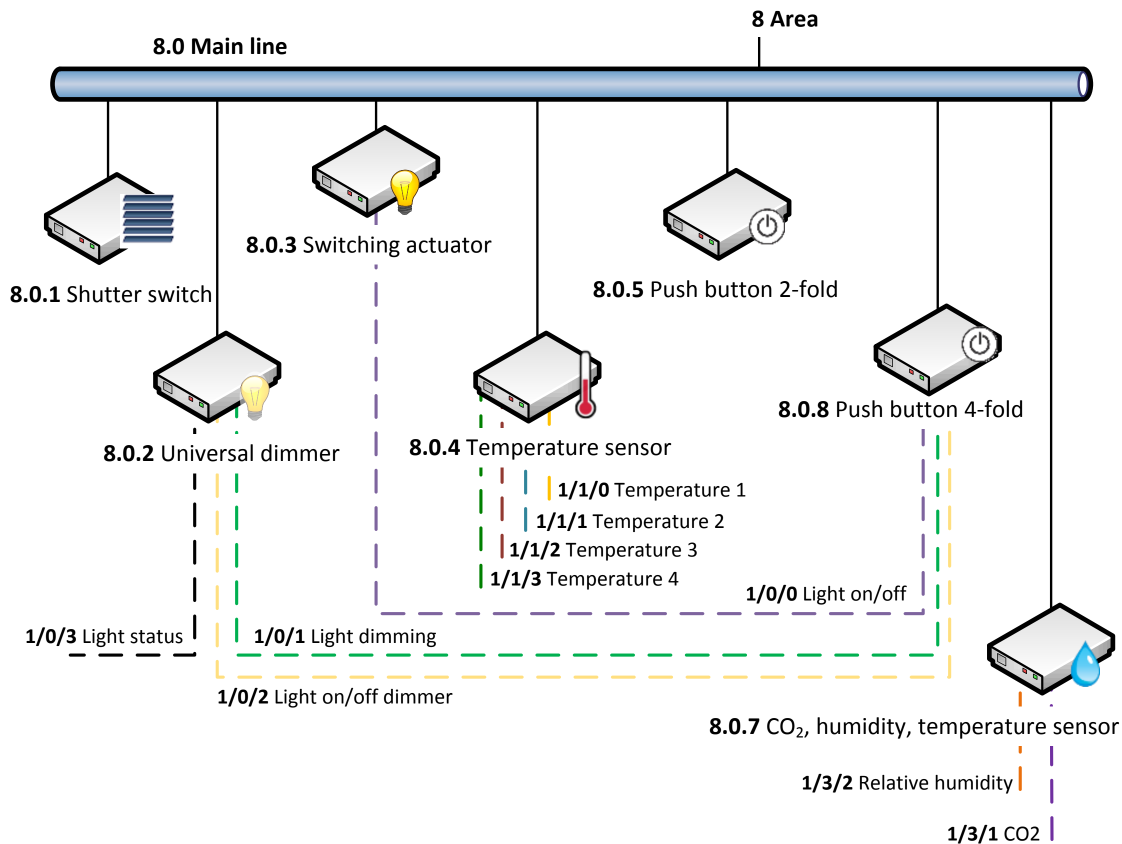

Figure 6.4: KNX evaluation network

The last part of the experimental setup is the connection between the KNX network and the oBIX gateway. As the network contains a KNXnet/IP router, this connection can be set up easily by using the Internet Protocol (IP). In contrast, the configuration of the devices is done by downloading the ETS4 settings via the network's USB interface.

6.1.3 Mapping

Before the model-driven approach can be evaluated, a procedure for mapping KNX networks to the MDA approach respectively to oBIX has to be defined. The preceding chapters pointed out the concepts of KNX networks and their connection to this model-driven approach. In this context, the conversion of KNX datapoints to oBIX objects is focused while the mapping of other network parts (e.g. topology structure, building view, list of devices) is straightforward to a certain degree.

The KNX specification defines a large set of datapoint types for a wide ranging field of applications. The types differ in their format (e.g. bit length), the unit (e.g. meter, kilogram), the allowed range of values (e.g. 0 to 127) and the encoding (e.g. *false* means *open*, *true* means *closed*) [38]. Within the scope of this thesis is the mapping of these types from KNX to oBIX.

DPT_ID	Format	DPT_Name	Implementation note				
			Value	Encoding	Range	Unit	Operations
1.001	B1	DPT_Switch	bool	enumeration			
1.002	B1	DPT_Bool					
1.003	B1	DPT_Enable					
2.001	B2	DPT_Switch_Control	bool (control, value)	enumerations: control (control, no control) value: see DPT-1			
3.007	B1U3	DPT_Control_Dimming	parameter: int (step code)	binary encoded	-100-100	%	increase, decrease
3.008	B1U3	DPT_Control_Blinds					up, down
4.001	A8	DPT_Char_ASCII	str	ASCII encoded	1-1		
4.002	A8	DPT_Char_8859_1		8859-1 encoded	1-1		
5.001	U8	DPT_Scaling	real	binary encoded	0-100	%	
5.003	U8	DPT_Angle			0-360	°	
6.001	V8	DPT_Percent_V8	int	two's complement notation	-128-127	%	
7.001	U16	DPT_Value_2_Ucount	int	binary encoded	0-65535	pulses	
7.002	U16	DPT_TimePeriodMsec	reltime			ms	
9.001	F16	DPT_Value_Temp	real	float value	-273-670760	°C	
9.002	F16	DPT_Value_Tempd			-670760-670760	K	
9.003	F16	DPT_Value_Tempa				K/h	
9.004	F16	DPT_Value_Lux			0-670760	Lux	
9.005	F16	DPT_Value_Wsp				m/s	
9.006	F16	DPT_Value_Pres				Pa	
9.007	F16	DPT_Value_Humidity				%	
9.008	F16	DPT_Value_AirQuality				ppm	
9.010	F16	DPT_Value_Time1			-670760-670760	s	
9.011	F16	DPT_Value_Time2				ms	

Figure 6.5: KNX datapoint mapping

Figure 6.5 provides an excerpt of this mapping in the form of a spreadsheet. The dark green colored fields indicate those datapoint types that occur in the experimental KNX network. For instance, the datapoint type 1.001 becomes an oBIX `Bool` element with an additional enumeration for the encoding of the binary value (*DPT_Switch*). The datapoint type 9.001 represents a 16-bit floating-point number which is mapped to a `Real` object with degree Celsius as unit and an accepted range of values of -273 to 670760 (*DPT_Value_Temp*). A special case is datapoint type 3.007 which is implemented by means of two operations for increasing and decreasing the brightness value (*DPT_Control_Dimming*). The operations take an `Int` object as input parameter representing the relative change in value.

Table 6.2 lists all configured datapoints of the KNX sample network. The datapoints are grouped per device. It should be noted that devices with not even one engineered datapoint are left out in this table. In the rightmost column, the KNX datapoint type is given. The implementation details of these types have already been discussed in the preceding paragraph or can be seen in Figure 6.5.

Device	Datapoint	Datapoint type
Universal dimmer	Switch, status	1.001
	Dimming	3.007
	Status	5.001
Switching actuator	Switch, channel A	1.001
Temperature sensor	Temperature, channel A	9.001
	Temperature, channel B	9.001
	Temperature, channel C	9.001
	Temperature, channel D	9.001
CO_2, humidity, temperature sensor	CO_2 value	9.008
	Rel. humidity value	5.001
	Threshold 1 CO_2	1.001
	Threshold 2 CO_2	1.001
	Threshold 3 CO_2	1.001
Push button 4-fold	Switch centre right	1.001
	Dimming on / off outer right	1.001
	Dimming outer right	3.007

Table 6.2: KNX datapoints

6.1.4 Evaluation

The evaluation of the developed, model-driven approach starts with the export of the KNX network's engineering data from the ETS4. The output of this export procedure is an archive file containing several files and folders. The aim is to transform this exported information into a model conforming to the BAS metamodel. Thus, this model will represent the PIM. As known from the transformation process, this step is called *network modeling*. Here, the automatic approach is chosen by implementing an XSLT stylesheet which results in the desired model file during execution.

First, the structure of the export archive is explained. Therefore, Figure 6.6 visualizes the particular components of this archive. Files that are not relevant for this modeling step are omitted in the description and the corresponding figure. In the root folder, the master file (*knx_master.xml*), which contains meta information about manufacturers, datapoint types and media types, is located. The project files are contained in a separate folder named according to the unique project identifier (*P-0341*). Within this project folder, the file *Project.xml* contains the name of the KNX project (i.e. the network name). The other network specific information is listed in the second file of this folder (*0.xml*). The used devices, their topology, the categorization into building parts, the group addresses, and so on are included in this kind of main file.

In addition, the archive contains folders for each manufacturer, if devices of this manufacturer are part of the KNX network. In this case, devices of two manufacturers are assembled in the network (Siemens, Schneider Electric). The folders are named after the unique manufacturer identifiers which are listed in the master file. Each folder contains a file called *Hardware.xml* that specifies the names, descriptions and order numbers of the installed devices. In the resulting

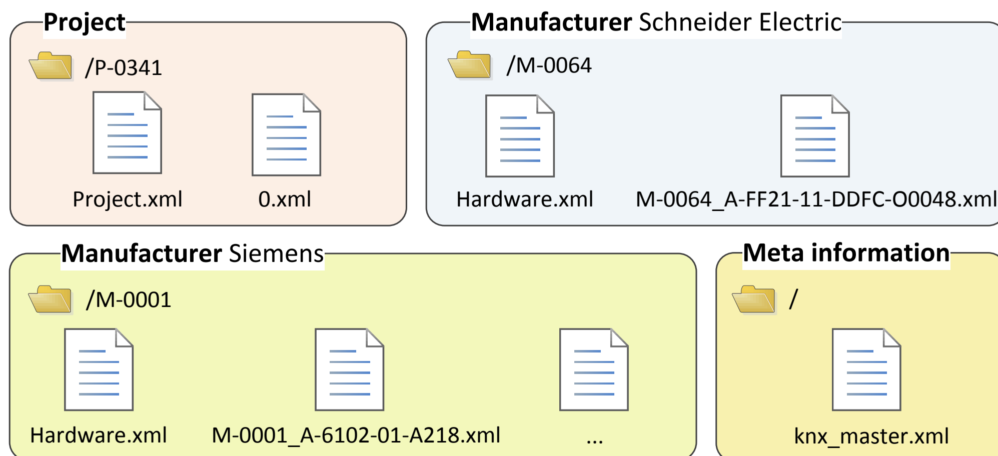

Figure 6.6: ETS4 export structure

PIM, this information is mapped to the attributes of an entity. Moreover, the manufacturer folder comprises one XML file per application program. Datapoints and parameters are specified in such a file. The parameters are excluded from mapping to the PIM as they are not part of the defined BAS modeling language. However, the datapoint information is used to create the datapoint elements in the BAS model. All files in the export archive contain multilingual texts that are transformed into translations of the BAS model.

Based on the main file of the project (*0.xml*), an XSLT stylesheet is executed. If required, the other files are linked within this transformation. The result of the network modeling phase is a model which conforms to the BAS metamodel. The other steps of the transformation workflow (model transformation, code generation) have been discussed in an earlier chapter (see Chapter 4). Hence, they are skipped and a generated, executable source code is assumed as this evaluation is more focused on the funtionality of the running system than on the step by step execution of the transformation process. The generated code instantiates the network including all its components in order to create an abstract representation of the BAS in the oBIX gateway.

During the initialization process of the gateway, the generated device loader class is executed. Finally, all oBIX objects, which are necessary to manage the BA network, have been instantiated in the gateway. The IoTSyS implementation provides a so called *lobby* where the available top level elements are listed. The HTTP call to get the lobby as well as the received response are shown in Listing 6.2. The lobby offers an entry point to additional meta objects like enumerations (`/enums`), units (`/units`), parameters (`/parameters`) and encodings (`/encodings`). These metadata are not part of the generated source code, but are instantiated before the gateway executes the configured device loaders.

One of the entries in the oBIX lobby refers to the KNX evaluation network which can be accessed via the server absolute URI `/networks/office`. Listing 3.7 shows the resulting object that is returned in response to calling this URI. The subobjects of the network conform to the descriptions and definitions of Section 3.4, as well.

```
GET http://localhost:8080/obix

-- response --
200 OK
Content-Type: text/xml
Date: Wed, 2 Apr 2014 20:01:44 GMT
Content-Length: 406

<obj href="obix/">
  <ref name="about" href="obix/about"/>
  <ref name="enums" href="enums"/>
  <ref name="units" href="units"/>
  <ref name="parameters" href="parameters"/>
  <ref name="encodings" href="encodings"/>
  <ref href="watchService" is="obix:WatchService"/>
  <ref href="alarms" is="obix:AlarmSubject"/>
  <ref name="P-0341" href="networks/office" is="bas:Network"
      displayName="Office"/>
</obj>
```

Listing 6.2: Requesting oBIX lobby

Next, some exemplary HTTP calls are examined to demonstrate the functionality of the generated source code, which is the result of the model-driven approach. First, a simple `GET` of a KNX datapoint is performed. Then, requests via `POST` and `PUT` are discussed. These three methods constitute the main part of the REST interface offered by the oBIX gateway. Although the examples are not able to cover all possible scenarios, they should illustrate the correct integration of the BAS model into the WS interface to manage the underlying BA network. As a result, Hypothesis 1 and 2 can be seen as confirmed.

```
GET http://localhost:8080/networks/office/datapoints/switch_channel_a/1/

-- response --
200 OK
Content-Type: text/xml
Date: Fri, 21 Mar 2014 10:05:47 GMT
Content-Length: 393

<obj name="P-0341-0_DI-3_M-0001_A-9803-03-3F77_O-3_R-4" href="/networks/
    office/datapoints/switch_channel_a/1/" is="knx:DPST-1-1 knx:DPT-1
    knx:Datapoint" display="On / Off" displayName="Switch, Channel A">
  <bool name="value" href="value" val="false" null="true" writable="true"/>
  <enum name="encoding" href="encoding" val="off" null="true" writable="true"
      range="/encodings/onoff"/>
</obj>
```

Listing 6.3: Reading a datapoint

A `GET` call via HTTP needs the URI of the targeted object. Additionally, parameters like the desired language can be added to the method call. In this example, the switch channel datapoint

(`switch_channel_a`) of the switching actuator will be read. As it was already stated earlier, the list of datapoints is a subobject of the network object and contains all available datapoints. The required object can be found by browsing this tree of oBIX objects, if the actual URI is not known. After sending the `GET` call, the answer of the server contains the relevant oBIX object.

The request and the entire response are shown in Listing 6.3. Herein, the content is returned as plain XML data, but the IoTSyS gateway offers a set of other encodings like JavaScript Object Notation (JSON) or Efficient XML Interchange (EXI) as well. As stated in Section 3.4.1, renaming the oBIX attribute `href` to `uri` is necessary due to naming conflicts in the XMI serialization of the PSM. In the oBIX gateway implementation, this attribute is labeled `href` again. The requested datapoint is of KNX type *DPST-1-1*. Therefore, it possesses a `Bool` property and an `Enum` property for the encoding of the binary value. In this example, the attribute `null` was set to *true* for both properties as the gateway has not captured the actual value of the underlying datapoint since system startup.

Next, an HTTP `PUT` call is performed with the intention to change the binary value of the same datapoint. If an oBIX object is specified as writable, it can be modified. Otherwise, a write attempt will achieve no result. In the sample network, the switching actuator is connected with a bulb. By setting the property `value` to *true*, the light will be switched on. In contrast to `GET`, `PUT` additionally sends the altered oBIX object in the content of the call. This content object replaces the outdated server object. The modified object is returned in the HTTP response. Listing 6.4 shows the raw transaction of this `PUT` call.

```
1  PUT http://localhost:8080/networks/office/datapoints/switch_channel_a/1/value
2  accept-language: en
3  Content-Type: text/xml
4  <bool name="value" val="true" null="true" writable="true" href="/networks/
       office/datapoints/switch_channel_a/1/value/"/>
5
6  -- response --
7  200 OK
8  Content-Type:  text/xml
9  Date:  Fri, 21 Mar 2014 12:20:19 GMT
10 Content-Length:  62
11
12 <bool name="value" href="value/" val="true" writable="true"/>
```

Listing 6.4: Modifying a datapoint

Not the whole datapoint object, but only the binary property is modified in this example. The server receives the new object and updates the existing one. Here, the new value is written to the KNX bus, as well. Consequently, the bulb is switched on. In order to check communication, the traffic on the KNX bus can be monitored by means of the ETS4.

Finally, a `POST` method is called to increase the value of the dimming actuator. The content of this call contains an oBIX object. In this case, the object is only a parameter for the invoked oBIX operation that is located behind the targeted URI. Listing 6.5 shows the HTTP transaction including the parameter object. The dimming value is incremented by 25 percent, and an appropriate message is sent to the actuator via the KNX bus while the operation is executed. In

the end, the content of the response comprises the operation's output object, if specified. For a better overview, unnecessary line breaks have been removed in the listings of this section.

```
POST http://localhost:8080/networks/office/datapoints/dimming/1/increase
accept-language: de
Content-Type: text/xml
<obj is="knx:ParameterDimming">
  <int name="value" val="25" unit="/units/percent"/>
</obj>

-- response --
200 OK
Content-Type:  text/xml
Date:  Fri, 21 Mar 2014 12:26:38 GMT
Content-Length:  33

<obj is="obix:Nil" null="true"/>
```

Listing 6.5: Invoking an operation

Besides HTTP, the CoAP can be used to interact with the oBIX gateway. One valuable feature of this protocol is the *observation* of individual URIs with respect to changes. When datapoints (e.g. the channel of a temperature sensor) send update information in periodic intervals over the KNX bus, the oBIX gateway recognizes these messages and informs registered CoAP clients. These clients do not need to poll the required data.

Here, the first use case can be derived. The observation can be helpful for a BMS observing the compliance of various threshold values in a building. CoAP minimizes the data traffic between the remote management software and the oBIX server. The software developer knows the simple interface and the protocols for communicating with this interface, but additional knowledge of the underlying BAS technology is not required.

Another example is the increased involvement of smartphones and other mobile devices as remote control for smart buildings. Any assembled BA technology can be seamlessly integrated in Web gateways. Thus, the mobile applications can be decoupled from proprietary software, and fully customized tools can be developed.

Independent of its technology (e.g. oBIX), a remote BMS is not limited to one distinct gateway. In fact, many BASs in different buildings can be combined in a large BMS even if the networks are integrated in different gateway servers. The monitoring and managing procedures of devices and datapoints in these networks are still the same. The developer simply has to know how a distinct element can be addressed.

All in all, the slight Web interface in combination with the univeral model-driven approach offers a powerful way for integrating BASs into the Internet in order to enable remote access via fully customizable BMSs.

6.2 Related work

This section covers already existing work related to the topics of this thesis. Approaches for integrating BASs into Web technologies are taken into account. Moreover, possible model-based and model-driven processes for this field of application are examined.

First, some integration approaches are discussed that do not use model-driven principles, but focus on the theoretical mapping concepts from BAS technologies to Web technologies. Neugschwandtner et al. report about the possibilities of integrating KNX into oBIX [48]. They identified Web services as key elements to enable interoperability between BASs, and oBIX with its clear REST interface is chosen as target platform. On this basis, another approach to access KNX via BACnet/WS is developed [34]. The singularities of BACnet/WS are pointed out and are compared with other related technologies (OPC UA, oBIX). Mapping options for a KNX integration are supplemented by a sketch of a proof of concept implementation. The integration of KNX into OPC UA by means of the ETS4 network data is subject of the research work of Fernbach et al. [22]. Besides the theoretical mapping, this work addresses a concrete implementation demonstrating the approach. Although these articles show the fundamental principles for mapping BAS technologies to Web technologies, they solely use manual mappings. Nonetheless, basic considerations of these approaches have inspired the evolution of the model-driven approach in this thesis.

The European Telecommunications Standards Institute (ETSI) has published standards concerning machine-to-machine communication. One of these is *ETSI TR 102 966* [20] which addresses the interoperability of ETSI machine-to-machine architecture and various network technologies like ZigBee or KNX. The standard contains the principles for mapping a BA network to the ETSI concept. Besides a general part, the report comprises mapping examples for oBIX. For instance, KNX datapoint type mappings and the network representation are examined. Herein, oBIX operations are used to enable the creation and modification of the network and its components during runtime.

Wang et al. show a middleware based on WSs for the integration of BASs [65]. Two main issues are discussed in this paper. First, the linkage of heterogeneous BASs is addressed. Second, the connection between these BASs and the Internet is brought up. In the local area network (LAN), the OPC technology is used as communication interface while remote BMSs located on a separate Web server utilize WS protocols like the SOAP to access BA components. BA technologies supporting OPC can be connected with the middleware's OPC interface. Therefore, the middleware bridges the gap between the local BA networks and the Internet.

In contrast, the literature also offers related work that is focused on model-driven approaches in the field of BASs. Muñoz et al. describe such a model-driven method [46]. Their MDA approach is used to build models that are automatically transformed into the final system. The three main components are a modeling language called *PervML*, an implementation framework based on OSGi and a transformation engine for the generation of Java code. While system analysts model the requirements in the form of PervML models, the system architect concretizes these abstract models by defining devices and systems. Technology specific knowledge is only required during the development of device drivers by OSGi developers. These drivers are linked with the generated code to form an executable application. Thereby, the approach is mainly

independent of technology specific issues. Nonetheless, Sánchez et al. argue that the abstract notation for the modeling of requirements is not that easy for BA experts [59].

Nain et al. present a middleware for BA built by an MDA approach [47]. This middleware called *EnTiMid* provides several service access models respectively personalities, and hence it is called schizophrenic. SOAP, UPnP and DPWS are examples for possible personalities. The architecture consists of three layers. On the lowest level, the communication with the physical devices is handled. The middleware EnTiMid is located in the middle layer, and on top high-level services are published to access the devices by means of standard protocols. The MDE approach is used to generate various personalities. Thus, an EnTiMid model representing the underlying sensors and actuators is mapped to one of the high-level service access models. Moreover, actual services are generated from abstract service descriptions. The complexity of different communication protocols on the physical layer is hidden by this layered middleware.

Yu et al. present another service-oriented and model-driven approach [67]. On the basis of the service-oriented computing (SOC) paradigm, a metamodel is developed to model services and service descriptions. A PIM, which conforms to this metamodel, represents the specific domain model. Model transformations are used to convert this PIM into two different models. One constists of the service elements (service composition model) while the other model includes the application specifications (application model). Both are still platform independent representations. The executable application is generated as PSM from the application model. In summary, this approach targets the modeling of complex business logic as opposed to the model-driven integration of BASs into a slim and common interface. Two years before, Bourcier et al. have dealt with the development of service-oriented gateways in order to control BA networks, as well [11].

All presented, model-driven approaches have the main focus regarding high-level services and business logic running on a middleware platform in common. The programming of the physical devices is still a task of experts. However, Sánchez et al. show an approach for generating code to program devices automatically [59]. Hence, no specific knowledge of the BA technology is needed. A DSL helps developers with low experience in the field of creating BASs to model such systems. Afterwards, transformations generate executable program code. As an example, the authors present the generation of a VBScript that is executed in the Engineering Tool Software (ETS) to configure a KNX network. The metamodel of this MDE approach includes the concept of a requirements catalog. Thus, modeled requirements can be reused in successive modeling processes. The requirements catalog is connected with a set of DSL fragments that form the resulting application model. Finally, the mentionend transformations generate code for specific BA platforms (e.g. ETS). Similar to the other presented approaches, the target technology can be easily exchanged without modifying the abstract application model. Hence, solely the platform specific code generation has to be developed.

This thesis is mainly concerned with the integration of already engineered BASs. The aim is to provide simple services for managing the network. Therefore, constructive high-level services are shifted to subsequent BMSs which are not in the focus of this approach. Likewise, the generation of network configuration code as stated by Sánchez et al. [59] has not been contemplated. The model-driven approach targets the precise mapping of the physical BA network and the automatic transformation of this representation to integrate the system into a WS gateway.

6.3 Open topics

In this section, some notes on disregarded and open topics are collected. The scope of the thesis is to develop a universal, model-driven approach for the integration of BASs. The principles of such an approach are presented, but the implementation makes no claim to be complete. The possibility to realize such an approach conforming to the MDA initiative has been shown taking the hypotheses introduced in Chapter 1 into account. Nonetheless, some open topics need to be mentioned at this point.

First, the BAS metamodel has been developed partly on the basis of KNX. Moreover, the case study uses only a KNX network to test the implementation of the model architecture and the transformation workflow. Hence, it is necessary to evaluate whether the approach is in conformity with other technologies like BACnet or LonWorks. On the one hand, the suitability of the already implemented concepts must be reconsidered. Particularly, the datapoint modeling is strongly influenced by the KNX approach. On the other hand, additional modeling elements have to be specified that are inspired by other BAS technologies. For example, new views may need to be introduced. The aim is to enable the model-driven integration approach for a wide range of BASs.

Second, alternative WSs-based technologies can be supported by the transformation workflow. This way, system engineers are not restricted to oBIX, but are able to generate code for various platforms to integrate the modeled BA networks. Possible solutions can be BACnet/WS or OPC UA (see Section 2.2). For this purpose, additional modeling languages in the form of metamodels have to be defined. Moreover, the model transformation from the PIM to these PSM needs to be implemented. This extension will increase the range of applicable target platforms, and will result in an increased suitability of the entire model-driven approach.

Last but not least, this implementation is more focused on the transformation of BA networks than on the mapping of metadata. Although the metamodels provide modeling concepts for meta information and some library models have been developed as PIMs, the model transformation and the subsequent code generation are not concerned with these libraries. With respect to a preferably complete implementation, meta information should be included in all steps of the transformation workflow.

Section 7.2 describes continuative issues regarding this field of application and beyond.

CHAPTER 7

Conclusion

7.1 Summary

With this thesis, an approach to integrate BASs into the IoT has been presented. Therefore, model-driven methodology and BAS technology independent interfaces have been used to realize the intended concepts. The most important aspects of this approach are summed up in the following paragraphs.

First, an appropriate architecture for the model-driven approach is introduced. OMG's MDA with its meta-metamodel MOF offers the required framework. Within the conventional four-layer architecture of MDA, two modeling languages are defined. Each of these modeling languages is specified by means of a metamodel which conforms to the MOF meta-metamodel. Thus, the meta-metamodel can be seen as metamodeling language for the definition of particular modeling languages. The metamodels determine the available language concepts for the creation of models that represent (real) systems. On the one hand, the BAS metamodel is used to generate models of BA networks in a technology and platform independent way. These models are called PIMs. On the other hand, the oBIX metamodel enables for the definition of models according to the oBIX technology that is chosen as target platform. Due to their relationship to a specific technology, these models are called PSMs. While PIMs provide a rather abstract perspective on the BA network and its components, the PSMs are expressed in terms of the underlying technology. The advantage of the layered MDA architecture and its common meta-metamodel is the interoperability among the models of different modeling languages.

Next, transformations are established with the aim to realize an automated workflow from a BA network model to the final source code executed in the Web interface technology. Three steps are identified in this transformation process. Foremost, the network has to be modeled either manually or automatically (*network modeling*). As soon as the network is available in the form of a PIM, the actual MDA workflow starts. An M2M transformation from the PIM to the PSM is established to map the abstract information to oBIX conform constructs (*model transformation*). In the last step, the program code is generated based on the PSM by means of a template-oriented M2T transformation approach (*code generation*).

These theoretical considerations become the corner stones of a proof of concept implementation. The realization of the model-driven approach uses the MDA implementations and packages of the Eclipse IDE. The entire implementation is split into two Eclipse workspaces containing various projects to provide a modular structure. Each step of the transformation process is seen as a separate project. Similarly, the metamodels and the models are combined in individual projects. In addition, libraries exist to support the transformations and metamodels. The EMF is used to create the mentioned metamodels as well as the associated model editors. Semantic constraints are integrated into the metamodels via the Eclipse OCL Tools, and the M2M transformation is realized by means of the QVT Operational extension. In order to generate source code, the Xpand package is utilized. As Java dominates the whole development process, the Java based IoTSyS integration middleware is chosen as oBIX implementation. Hence, the final source code can be executed directly on this target platform.

The evaluation of the proof of concept implementation consists of a case study based on an experimental KNX network. This network contains several devices like a temperature sensor, push buttons, a switching actuator or a dimming actuator. The components and their interworking are engineered in the ETS4. An export of the engineering data is used for further processing. Initially, the KNX information is mapped to the BAS metamodel by generating a PIM. After running the MDA transformations, the source code representing the KNX network is executed within the IoTSyS gateway. Sample interactions with the oBIX implementation illustrate the functional capability of the implementation and the entire model-driven approach.

7.2 Future work

The presented approach offers a basis for further development with respect to a seamless and complete integration of BASs into the upcoming IoT. Open topics mentioned in Section 6.3 already give an overview of possible short-term adaptions of the taken approach. Additionally, this section points out some long-term issues.

One open topic is the validation of the BAS model with other BA technologies. Furthermore, the support of multiple target platforms in addition to oBIX and the code generation for all network elements and libraries need to be resolved. These measures will enhance the applicability of the model-driven approach. Hence, it will be possible to map the most common BA technologies to a set of WSs-based integration technologies.

Currently, the workflow is unidirectional, and the entry point is the already engineered BAS. After some transformation steps, the final source code is generated. A possible modification could be the definition of a bidirectional transformation process. Given the source code or the corresponding PSM that represents a BAS by means of a specific technology, the PIM should be generated via MDA transformations. Finally, the aim is to produce network configuration data on the basis of the PIM.

The long term goal behind this work is that the abstract model of a BA network becomes the most important element in the integration workflow. Both the generation of network configuration data and the gateway source code are part of subsequent process steps. The workflow from the BA network to the source code and vice versa becomes a BAS model-centric process. Thus, no technology dependency will remain at all. A network integrator only relies on the abstract

network representation. Both target technologies, in the network's field level and in the IoT interface, can be easily replaced without any changes to structure and semantics of the modeled network.

In addition, the ability of ontologies to create new knowledge by reasoning should be taken into account. In Section 2.1.6, an approach for integrating ontologies into the MDA concept has been presented. The ontology models can be used as intermediaries between the current PIM and PSM in order to influence the transformation definitions. New knowledge can be generated based on collected information to modify the transformation definitions. Thus, they can be customized to fit to the changed needs of the BA technologies respectively the remote users behind the IoT interface.

An overall integration methodology for both legacy systems and new BASs is able to vastly reduce the necessary development time. Otherwise, the engineering of the BA network as well as the integration of this network into a gateway technology has to be done either manually or by various incompatible tools. Further research on efficiency and economy of time will support this assumption.

List of Acronyms

API Application programming interface.

ASHRAE American Society of Heating, Refrigerating and Air Conditioning Engineers.

ATL ATLAS Transformation Language.

BA Building automation.

BACnet Building Automation and Control Networks.

BACnet/WS Web Services for Building Automation and Control Networks.

BAS Building automation system.

BMS Building management system.

CASE Computer-Aided Software Engineering.

CIM Computation independent model.

CMOF Complete MOF.

CoAP Constrained Application Protocol.

COM Component Object Model.

DCOM Distributed COM.

DPWS Devices Profile for Web Services.

DSL Domain specific language.

E-Mode Easy Mode.

EMF Eclipse Modeling Framework.

EMOF Essential MOF.

ETS Engineering Tool Software.

ETS4 Engineering Tool Software 4.

ETSI European Telecommunications Standards Institute.

EXI Efficient XML Interchange.

GPL General-purpose modeling language.

HTTP Hypertext Transfer Protocol.

HVAC Heating, ventilation and air conditioning.

I/O Input/output.

IDE Integrated development environment.

IoT Internet of Things.

IP Internet Protocol.

IT Information technology.

JDK Java Development Kit.

JSON JavaScript Object Notation.

LAN Local area network.

LHS Left hand side.

M2M Model-to-model.

M2T Model-to-text.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

MOF Meta Object Facility.

Mof2Text MOF Model to Text Transformation Language.

MWE Modeling Workflow Engine.

OASIS Organization for the Advancement of Structured Information Standards.

oAW OpenArchitectureWare.

oBIX Open Building Information Exchange.

OCL Object Constraint Language.

ODM Ontology Definition Metamodel.

OMG Object Management Group.

OPC OLE for Process Control.

OPC UA OPC Unified Architecture.

OSGi Open Services Gateway initiative.

OWL Web Ontology Language.

PIM Platform independent model.

PSM Platform specific model.

QVT Query/View/Transformation.

QVTO QVT Operational.

RDF Schema Resource Description Framework Schema.

REST Representational State Transfer.

RFID Radio-frequency identification.

RHS Right hand side.

S-Mode System Mode.

SI International System of Units.

SOA Service-oriented architecture.

SOAP Simple Object Access Protocol.

SOC Service-oriented computing.

TP Twisted pair.

UML Unified Modeling Language.

UPnP Universal Plug and Play.

URI Uniform Resource Identifier.

USB Universal Serial Bus.

WS Web service.

WWW World Wide Web.

XMI XML Metadata Interchange.

XML Extensible Markup Language.

XSLT Extensible Stylesheet Language Transformation.

List of Figures

1.1 Overview of the model-driven approach 4

2.1 Basic notions in MDE [8, 9] 8
2.2 MDA process [7] 9
2.3 Bridging RDF Schema and MOF [26] 15
2.4 oBIX object model [49] 17

3.1 Modeling stack [8] 22
3.2 EMOF core classes [52] 23
3.3 EMOF data types [52] 24
3.4 EMOF types [52] 24
3.5 EMOF packages [52] 25
3.6 Ecore kernel [63] 26
3.7 Sample automation network 28
3.8 BAS metamodel 31
3.9 BAS network model 36
3.10 oBIX metamodel 39

4.1 Transformation workflow 44
4.2 Model transformation from BAS to oBIX 46
4.3 Mapping of element and translation 48
4.4 Mapping of network 49
4.5 Mapping of datapoints view 50
4.6 Mapping of functional view 51
4.7 Mapping of entity 52
4.8 Mapping of datapoint 53
4.9 Mapping of example network 55
4.10 Mapping of example datapoint 56
4.11 Code generation procedure 58

5.1 Development environment and components 63
5.2 Xpand properties 64
5.3 IoTSyS packages 66
5.4 Workspace projects 68

5.5 Metamodel project structure 70
5.6 Ecore diagram editor 71
5.7 Metamodel library project structure 73
5.8 Model project structure 74
5.9 Model transformation project structure 75
5.10 Model transformation library project structure 77
5.11 Code generation project structure 79
5.12 Deployment sequence 83

6.1 KNX specific overview 87
6.2 ETS4 user interface 88
6.3 KNX testbed 90
6.4 KNX evaluation network 91
6.5 KNX datapoint mapping 92
6.6 ETS4 export structure 94

List of Listings

3.1 OCL invariant for enumeration literals . . . 32
3.2 OCL invariant for type hierarchy . . . 32
3.3 BAS unit model . . . 34
3.4 BAS enumeration model . . . 35
3.5 BAS type model . . . 35
3.6 BAS entity model . . . 36
3.7 oBIX network model . . . 40
3.8 oBIX entity model . . . 40
3.9 oBIX datapoint model . . . 41
4.1 Example of generated source code . . . 59
4.2 M2T template snippet . . . 60
5.1 KNX DPST-1-1 implementation snippet . . . 67
5.2 oBIX DateTime type snippet . . . 72
5.3 QVT header . . . 75
5.4 QVT network mapping . . . 76
5.5 Model transformation utility function . . . 78
5.6 Xpand main template definition . . . 80
5.7 Xpand template definition . . . 81
5.8 Model transformation settings . . . 83
5.9 Console output of M2T transformation . . . 84
6.1 oBIX gateway configuration . . . 90
6.2 Requesting oBIX lobby . . . 95
6.3 Reading a datapoint . . . 95
6.4 Modifying a datapoint . . . 96
6.5 Invoking an operation . . . 97

List of Tables

3.1 Ecore data type mapping [63] 26

4.1 Mapping of type properties 54

6.1 KNX devices 89
6.2 KNX datapoints 93

Bibliography

[1] About the Eclipse Foundation. `http://www.eclipse.org/org`. Accessed: 2014-04-15.

[2] ASHRAE. *A Data Communication Protocol for Building Automation and Control Networks (ANSI/ASHRAE Addendum c to ANSI/ASHRAE Standard 135-2004)*, September 2006.

[3] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36 – 41, September 2003.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, October 2010.

[5] BACnet Website. `http://www.bacnet.org`. Accessed: 2014-01-05.

[6] P.J. Barendrecht. Modeling transformations using QVT Operational Mappings. Research project report, Eindhoven University of Technology, April 2010.

[7] Steffen Becker. Coupled Model Transformations. In *Proceedings of the ACM International Workshop on Software and Performance*, pages 103 – 114, June 2008.

[8] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *Upgrade*, 5(2):21 – 24, April 2004.

[9] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171 – 188, May 2005.

[10] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 273 – 280, November 2001.

[11] Johann Bourcier, Antonin Chazalet, Mikaël Desertot, Clément Escoffier, and Cristina Marin. A Dynamic-SOA Home Control Gateway. In *Proceedings of the IEEE International Conference on Services Computing*, pages 463 – 470, September 2006.

[12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, September 2012.

[13] Calimero. http://sourceforge.net/projects/calimero. Accessed: 2014-01-04.

[14] Gonçalo Cândido, François Jammes, José Barata de Oliveira, and Armando W. Colombo. SOA at Device level in the Industrial domain: Assessment of OPC UA and DPWS specifications. In *Proceedings of the IEEE International Conference on Industrial Informatics*, pages 598 – 603, July 2010.

[15] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pages 1 – 17, October 2003.

[16] Keith Duddy, Anna Gerber, and Kerry Raymond. Eclipse Modeling Framework (EMF) import/export from MOF/JMI. Status report, Co-operative Centre for Enterprise Distributed Systems, May 2003.

[17] Eclipse Modeling - MMT. http://www.eclipse.org/mmt. Accessed: 2014-03-29.

[18] Eclipse Modeling Project. http://www.eclipse.org/modeling. Accessed: 2014-02-22.

[19] ETS4 Description. http://www.knx.org/knx-en/software/ets/about. Accessed: 2014-04-14.

[20] ETSI. *Machine-to-Machine communications (M2M); Interworking between the M2M Architecture and M2M Area Network technologies (ETSI TR 102 966 V1.1.1)*, February 2014.

[21] Andreas Fernbach. Interoperability at the Management Level of Building Automation Systems. Master's thesis, Vienna University of Technology, January 2013.

[22] Andreas Fernbach, Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. Mapping ETS4 Project Structure to OPC UA using ETS4 XML Export. In *KNX Scientific Conference*, November 2012.

[23] Peter Fettke and Peter Loos. Model Driven Architecture (MDA). *Wirtschaftsinformatik*, 45(5):555 – 559, October 2003.

[24] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[25] Flags - KNX/EIB - Lexikon - KNX-User-Forum. http://knx-user-forum.de/lexikon. Accessed: 2014-01-29.

[26] Dragan Gašević, Dragan Djurić, and Vladan Devedžiá. *Model Driven Engineering and Ontology Development*. Springer Berlin Heidelberg, 2nd edition, May 2009.

[27] Tom Hannelius, Mikko Salmenperä, and Seppo Kuikka. Roadmap to adopting OPC UA. In *Proceedings of the IEEE International Conference on Industrial Informatics*, pages 756 – 761, July 2008.

[28] Internet of Things. http://www.internet-of-things.eu. Accessed: 2014-01-05.

[29] IoTSyS - Internet of Things integration middleware. http://www.iotsys.org. Accessed: 2014-02-24.

[30] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1 – 2):31 – 39, June 2008.

[31] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128 – 138. Springer Berlin Heidelberg, 2006.

[32] Markus Jung, Jürgen Weidinger, Christian Reinisch, Wolfgang Kastner, Cedric Crettaz, Alex Olivieri, and Yann Bocchi. A Transparent IPv6 Multi-protocol Gateway to Integrate Building Automation Systems in the Internet of Things. In *Proceedings of the IEEE International Conference on Green Computing and Communications*, pages 225 – 233, November 2012.

[33] Wolfgang Kastner and Georg Neugschwandtner. Datenkommunikation in der verteilten Gebäudeautomation. *Bulletin SEV/AES*, 17:9 – 14, 2006.

[34] Wolfgang Kastner and Stefan Szucsich. Accessing KNX networks via BACnet/WS. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, pages 1315 – 1320, June 2011.

[35] Martin Kempa and Zoltán Ádám Mann. Model Driven Architecture. *Informatik-Spektrum*, 28(4):298 – 302, August 2005.

[36] KNX Association. http://www.knx.org/knx-en. Accessed: 2014-04-14.

[37] KNX Association. *KNX System Specifications, Architecture, Version 3.0*, June 2009.

[38] KNX Association. *KNX System Specifications, Interworking, Datapoint Types, Version 1.4*, April 2009.

[39] KNX Configuration Modes. http://www.knx.org/knx-en/knx/technology/configuration-modes. Accessed: 2014-04-14.

[40] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart Objects as Building Blocks for the Internet of Things. *IEEE Internet Computing*, 14(1):44 – 51, January 2010.

[41] LonWorks. http://www.echelon.de/technology/lonworks. Accessed: 2014-01-05.

[42] Boris Malinowsky, Georg Neugschwandtner, and Wolfgang Kastner. Calimero: Next Generation. In *KNX Scientific Conference*, November 2007.

[43] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 290 – 297. Springer Berlin Heidelberg, 2002.

[44] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, March 2006.

[45] Model-Driven Engineering for Building Automation Systems. `http://www.auto.tuwien.ac.at/~dschachinger/da`. Accessed: 2014-04-20.

[46] Javier Muñoz, Vicente Pelechano, and Carlos Cetina. Implementing a Pervasive Meeting Room: A Model Driven Approach. In *Proceedings of the International Workshop on Ubiquitous Computing*, pages 13 – 20, May 2006.

[47] Grégory Nain, Erwan Daubert, Olivier Barais, and Jean-Marc Jézéquel. Using MDE to Build a Schizophrenic Middleware for Home/Building Automation. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 49 – 61. Springer Berlin Heidelberg, 2008.

[48] Matthias Neugschwandtner, Georg Neugschwandtner, and Wolfgang Kastner. Web Services in Building Automation: Mapping KNX to oBIX. In *Proceedings of the IEEE International Conference on Industrial Informatics*, volume 1, pages 87 – 92, June 2007.

[49] OASIS. *OBIX Version 1.1, Committee Specification Draft 02 / Public Review Draft 02*, December 2013.

[50] Obix Java Toolkit. `http://sourceforge.net/projects/obix`. Accessed: 2014-02-15.

[51] OMG. *MDA Guide Version 1.0.1*, June 2003.

[52] OMG. *Meta Object Facility (MOF) 2.0 Core Specification, Version 2.0*, October 2003.

[53] OMG. *MOF Model to Text Transformation Language, Version 1.0*, January 2008.

[54] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, January 2011.

[55] openArchitectureware. `http://www.openarchitectureware.org`. Accessed: 2014-02-23.

[56] Roland Petrasch and Oliver Meimberg. *Model Driven Architecture: Eine praxisorientierte Einführung in die MDA*. dpunkt.verlag, June 2006.

[57] Iman Poernomo. The Meta-Object Facility Typed. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1845 – 1849, April 2006.

[58] Christian Reinisch, Wolfgang Granzer, Fritz Praus, and Wolfgang Kastner. Integration of Heterogeneous Building Automation Systems using Ontologies. In *Proceedings of the Conference of the IEEE Industrial Electronics Society*, pages 2736 – 2741, November 2008.

[59] Pedro Sánchez, Manuel Jiménez, Francisca Rosique, Bárbara Álvarez, and Andrés Iborra. A framework for developing home automation systems: From requirements to code. *Journal of Systems and Software*, 84(6):1008 – 1021, June 2011.

[60] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25 – 31, February 2006.

[61] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42 – 45, September 2003.

[62] Ayman Sleman and Reinhard Moeller. Integration of Wireless Sensor Network Services into other Home and Industrial networks using Device Profile for Web Services (DPWS). In *Proceedings of the International Conference on Information and Communication Technologies: From Theory to Applications*, pages 1 – 5, April 2008.

[63] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, December 2008.

[64] Valeriy Vyatkin. Software Engineering in Industrial Automation: State-of-the-Art Review. *IEEE Transactions on Industrial Informatics*, 9(3):1234 – 1249, August 2013.

[65] Shengwei Wang, Zhengyuan Xu, Jiannong Cao, and Jianping Zhang. A middleware for web service-enabled integration and interoperation of intelligent building systems. *Automation in Construction*, 16(1):112 – 121, January 2007.

[66] XML Schema Part 2: Datatypes Second Edition. `http://www.w3.org/TR/xmlschema-2`. Accessed: 2014-03-30.

[67] Jianqi Yu, Philippe Lalanda, and Stéphanie Chollet. Development Tool for Service-Oriented Applications in Smart Homes. In *Proceedings of the IEEE International Conference on Services Computing*, volume 2, pages 239 – 246, July 2008.