FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

model-UA:

Ein Open Source Tool zur Transformation von UML zu OPC UA Modellen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Sebastian Wiedemann
Matrikelnummer 1425647

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Thomas Frühwirth

Wien, 9. November 2018

Sebastian Wiedemann   Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

model-UA:

An open source tool for UML to OPC UA model transformations

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer engineering

by

Sebastian Wiedemann
Registration Number 1425647

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn Wolfgang Kastner
Assistance: Dipl.-Ing. Thomas Frühwirth

Vienna, 9th November, 2018

Sebastian Wiedemann   Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Sebastian Wiedemann
Vienna

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.

Wien, 9. November 2018

Sebastian Wiedemann

Kurzfassung

Die moderne Industrie fordert zunehmend die steigende Nachfrage nach cyber-physischen Systemen. Solche Systeme sind durch die hohe Komplexität, die durch den Verbund von vielen heterogenen Komponenten kommt, bekannt. Um dieser Herausforderung zu begegnen, gibt es Middleware-Technologien und standardisierte Ansätze wie beispielweise OPC Unified Architecture (OPC UA). Der Nachteil dabei ist, dass Modellieren mit dieser Modellierungssprache ein komplexer Vorgang ist. Zusätzlich gibt es kaum OPC UA Modellierungswerkzeuge, schon gar nicht in der open-source Community. Hier kommt Unified Modelling Language (UML) ins Spiel, ein alt-bekannter Modellierungs-Standard. Diese Arbeit präsentiert ein open-source UML Modellierungswerkzeug mit einer OPC UA-Erweiterung, um die Komplexität von OPC UA zu umgehen.

Abstract

The rising demand of Cyber-Physical Production Systems (CPPSs) and their need for the integration of Cyber-Physical Systems (CPSs) lead to a huge interoperability challenge. OPC UA tries to tackle this problem. It is a fairly new communication standard which allows to connect heterogeneous systems in a standardized way. The disadvantage of this, however, is that OPC UA modelling is inherently complex and that OPC UA modelling tools are scarce. Most importantly, there are no fully operational and actively worked on open-source projects for OPC UA modelling. This is where UML, a well-known modelling standard, comes in handy. This thesis presents an open-source UML modelling tool with an OPC UA modelling extension and a UML to OPC UA transformation approach to tackle the complexity of creating OPC UA information models.

Contents

CHAPTER 1

Introduction

Today's challenge in manufacturing is the transformation of common manufacturing systems into CPPSs. This presupposes the integration of CPSs into these systems. The usage of CPSs leads to a connection of sensors, machines and manufacturing systems. All these subsystems with their own interfaces and communication standards make interoperability between different components challenging. OPC UA, the latest specification of the OPC standard, accepts this challenge by allowing to connect heterogeneous systems in a standardized way. Even though the usage of OPC UA in this domain seems to be appealing, the problem with OPC UA is the lack of tools as well as the complexity of the model creation, implementation and support [1]. This thesis offers a way to tackle these problems by presenting an open source tool for UML with the ability to transform UML diagrams into OPC UA information models.

1.1 Motivation and problem statement

This thesis is mainly motivated by the lack of tools for OPC UA support. While there are some commercial products with full OPC UA functionality, the open-source community has not seen any significant development in OPC UA modelling tools, especially not actively worked on projects. Obviously, for the purpose of this thesis it would be a feasible idea to contribute to one of the existing open source modelling tools.

Furthermore, the usefulness of OPC UA comes with the price of high modelling complexity. This is where UML comes in handy. UML is a major established modelling language in the modelling community, especially for software development. Additionally, it is also used in the manufacturing environment. UML has various active open source modelling tool projects and there already are UML to OPC UA model transformation approaches available [1]. Subsequently, this thesis was born to modify an open source modelling tool in order to create OPC UA information models by transforming UML diagrams.

1.2 Model representation

A model is a simplified representation of the reality. A model can be physical like the mould of a steel pipe, a representation of a real life object like a building plan or theoretical like the hierarchical description of employers in a business. Modelling is the process to simplify and abstract objects and actions to gain a better overview of a system. There are different ways to describe a model. For instance, mathematical models describe systems using mathematical concepts, while information models use abstract depictions of objects in addition to their properties and relations between these objects to describe a model. Mostly, models are somehow drawn with specific modelling standards which describe the graphical notations. A metamodel is a model which describes the rules of another model. Additionally, there are often equivalent textual representations of graphical models. The process to get the textual representation of a graphical model is one aspect of model transformation. However, model transformation is also the process to transform one model standard into another. This is especially important for computer systems to exchange models.

While textual representation can in theory be as random as an essay, there are standardized ways to use them. Especially important are the ones for computers. One of them is Extensible Markup Language (XML), a language with a well-defined hierarchical structure. Additionally, there are some subtypes of XML like XML Metadata Interchange (XMI) which works the same way but has some additional rules.

1.3 Methodological approach and structure of this thesis

Firstly, this thesis provides some technical background in Chapter 2. In detail, this chapter contains general information about the two modelling languages (UML and OPC UA) used in this thesis, as well as theoretical background information about model transformations. After that, there are evaluations of existing work on the topic of UML to OPC UA model transformation in the state of the art (Chapter 3).

In Chapter 4, the explanation for the model transformation that was developed for this thesis is located. For the automated UML diagram to OPC UA information model transformation, an XMI file of a UML diagram is needed which is provided by Modelio 3.7, the UML modelling software used in this thesis. Basically, this is a text file which represents the UML diagram as text in an XML format. This file, as the starting point of the transformation, in combination with a specific transformation rule set leads to a OPC UA information model file in XML format. A small example is provided to show one of the possible transformations between the used modelling languages.

The thesis continues with the proof of concept (Chapter 5) which starts by presenting open source UML modelling tools with a special focus on Modelio 3.7. Furthermore, the OPC UA modeling extension for Modelio 3.7 is discussed in this chapter. This extension implements the UML-to-OPC UA Model-to-Model (M2M) transformation. Additionally, it is possible to create OPC UA information models using its graphical notation, export these models in their respected XML representation, and also import OPC UA XML files. This chapter concludes with the explanation of the algorithm used for the model transformation in Modelio. The thesis ends with a short summary and ideas on how to improve and extend the algorithm with additional features in Chapter 6.

CHAPTER 2

Technical background

This chapter contains the definitions and explanations of UML, OPC UA and model transformation. These are the technologies used in the approach presented in this thesis.

2.1 UML

UML is an industry standard for visualizing, specifying, constructing and documenting the artifacts of software systems. It has been established as the de facto standard modelling language. It allows and is mostly used to develop diagrams for Platform Independent Models (PIMs). UML is standardized by the Object Management Group (OMG) to unify object oriented methods [2, 3]. Figure 2.1 shows an excerpt of the UML metamodel

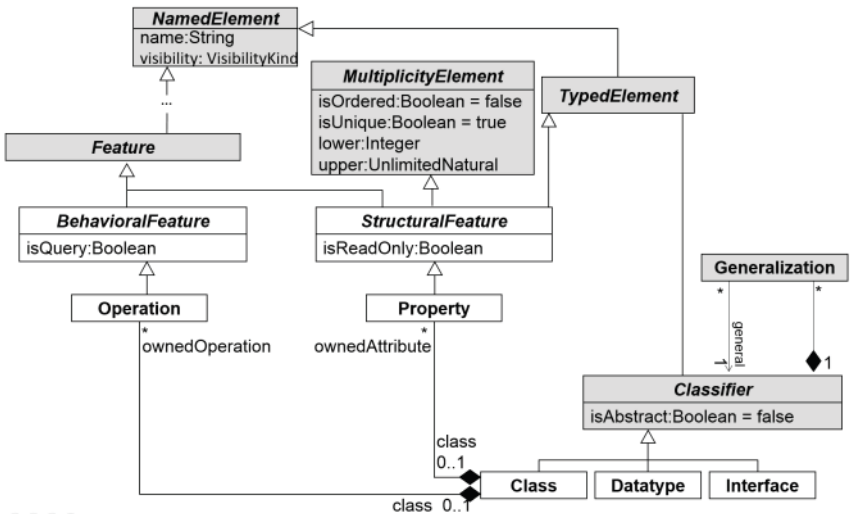


Figure 2.1: Metamodel of UML (excerpt)[1]

containing most elements which are relevant for the UML to OPC UA transformation. In the metamodel, every element is a subtype of *NamedElement*. Very important for the transformation idea are the relations between elements, especially the relation between a *Class* and *Property* (*ownedAttribute*) (e.g. private global class variable) as well as the relation between *Class* and *Operation* (*ownedOperation*) (e.g. a method of the class). In UML the relationships are directly defined in the metamodel.

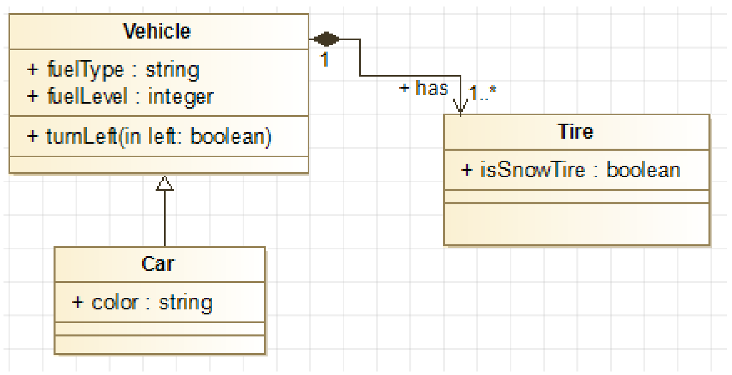


Figure 2.2: UML diagram example of a *Car*

Figure 2.2 provides a simple example of a UML diagram to showcase important UML modelling concepts. The diagram describes a *Car* generalized as *Vehicle* which is composed of at least one *Tire*. Furthermore, all classes have attributes which describe the classes more precisely. Operations provide classes with the ability for interactions.

2.2 OPC UA

OPC UA is a M2M communication protocol which is developed and standardized by the OPC Foundation. It is based on and extends Classic OPC. It was designed for the automation industry but can be applied to many more areas. An important goal of OPC UA is to achieve platform independence (see Computation Independent Model (CIM)/PIM).

OPC UA builds on different layers shown in Figure 2.3 with the transport mechanism and data modelling as fundamental components. OPC UA defines its own optimized binary Transmission Control Protocol (TCP) protocol but also a mapping to well known standards like XML and HTTP. The metamodel defines the rules and base building blocks to expose an information model with OPC UA. The OPC UA Services are an interface between the servers providing data and managing a system and the clients [4]. The metamodel is fundamental for this thesis. It also defines a graphical notation for modelling an OPC UA information model (see Figure 2.4).

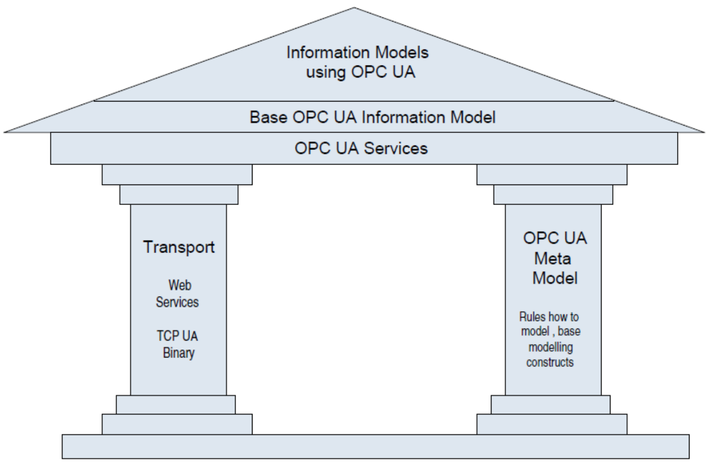
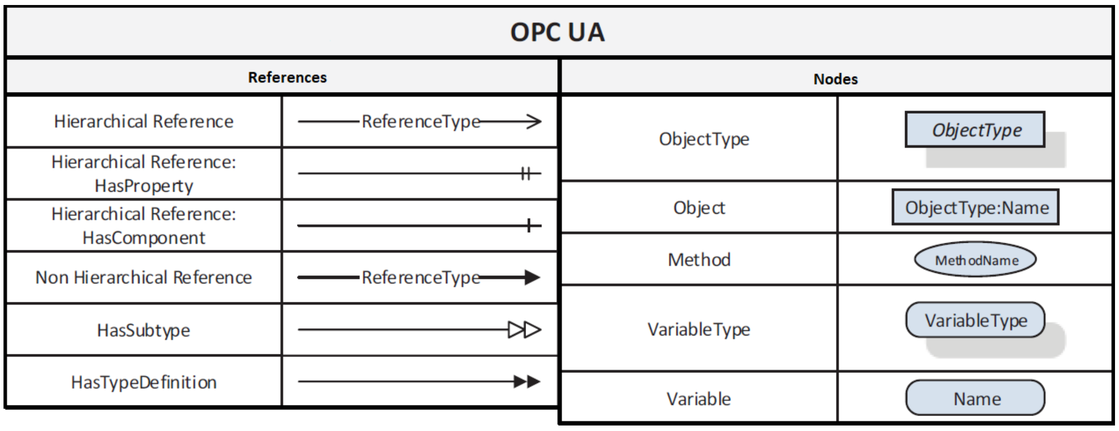


Figure 2.3: The foundation of OPC UA [4]



OPC UA			
References		**Nodes**	
Hierarchical Reference	—ReferenceType→	ObjectType	*ObjectType*
Hierarchical Reference: HasProperty	—‖	Object	ObjectType:Name
Hierarchical Reference: HasComponent	—+	Method	MethodName
Non Hierarchical Reference	—ReferenceType▶	VariableType	VariableType
HasSubtype	—▷▷	Variable	Name
HasTypeDefinition	—▶▶		

Figure 2.4: OPC UA graphical notation (adapted from [1])

Figure 2.5 shows the metamodel of OPC UA in *"UML-style"*. The OPC UA metamodel consists of nodes and references between them. In OPC UA, every node has a specific type (leaf nodes in the type tree in Figure 2.5). Each node has predefined type-specific attributes. Some of these, like the unique *NodeId*, are mandatory.

In OPC UA, relationships are defined as a list of references with a specific reference type [1]. This enables one of the most important abilities of OPC UA: adding new features to models without affecting existing ones. This can be seen in the metamodel. Adding features would simply result in more UANodes and more references between them. Nothing that previously existed is affected, wheras in UML for instance a new variable would affect an instanced class composition directly. Depending on the feature it may be necessary to change large portions of a UML design.

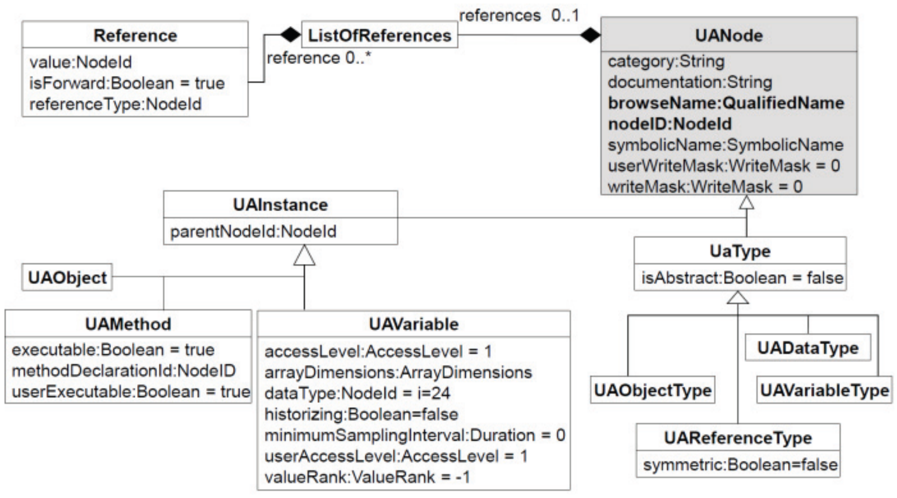


Figure 2.5: Metamodel of OPC UA (excerpt based on UANodeSet.xsd, http://opcfoundation.org/UA/schemas/1.02/UANodeSet.xsd)[1]

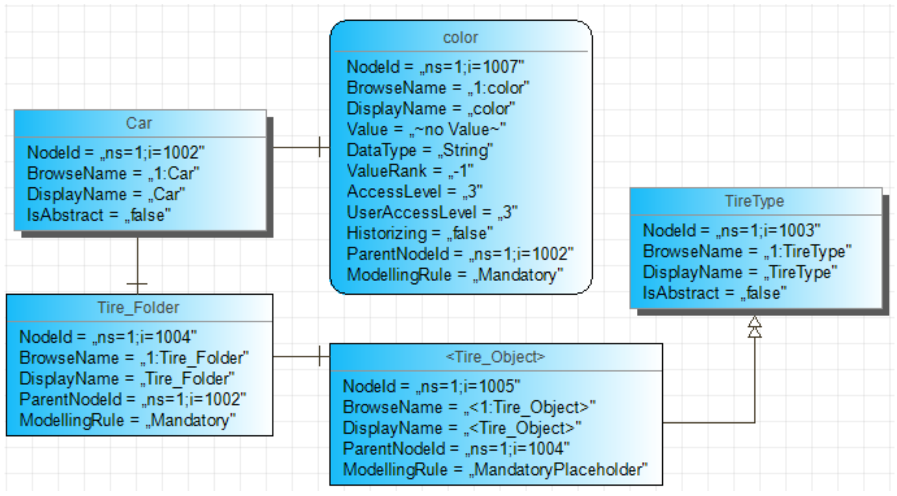


Figure 2.6: OPC UA information model example of a *Car*

Figure 2.6 provides a simple example OPC UA information model based on its UML counterpart from Figure 2.2. This example omits many attributes from the UML example, because OPC UA information models tend to get much bigger than their UML equivalent. The example shows a *Car* node which has a *Color* and is composed of at least one *Tire*. The composition is modeled via the *Tire_Folder* which manages *<Tire_Object>*s with a *"MandatoryPlaceholder"* modelling rule.

2.3 Model transformation

The goal of Model-Driven Engineering (MDE) is to tackle the increasing complexity of systems. With the principle *"everything is a model"*, MDE ensures the coherence of model-driven techniques just like the *"everything is an object"* principle helped object oriented methods to improve their techniques. In 2001, the OMG adopted Model-Driven Architecture (MDA). It supports MDE of software systems for the approach to use models in software engineering. MDA has three goals: portability, interoperability and reusability. A key technique of MDA is model transformation [1]. There are two important kinds of model transformations which are relevant for this thesis:

Model-to-Text (M2T) transformation

- a program takes a model as input and transforms it into text. This can result in a text file representing the model in e.g. XML or code generation (synthesizing systems).

M2M transformation

- a program takes a model as input and transforms it into another model.

Transformations are specified on a metamodel level. Therefore, it is possible to reuse a model transformation for all valid models conforming to the input metamodels.

CHAPTER 3

State of the art

3.1 A systematic approach to OPC UA information model design

Referring [5]

This paper presents an MDA approach using UML class, state-chart, use-case and component diagrams for virtual representation of manufacturing systems which allow an OPC UA information model design in the manufacturing domain.

UML is the proclaimed modelling language for MDA. Mostly because it is well known and often taught. UML also has an XMI representation which is used as an important standard for MDA.

Figure 3.1 shows an overview of the developed approach. The domain description (or CIM) is defined with UML diagrams. Component diagrams are used to define the static structure while use-case diagrams describe the functionality of the system. The CIM construction and the transformation to PIM are done manually and require the knowledge of domain experts and system users. The proposed approach uses object oriented modelling methods in addition to the information defined in the CIM to identify the UML classes and their related attributes, operations and relations between the classes. UML state machines are used to describe event-driven manufacturing systems. UML has some model elements and concepts which cannot be used in OPC UA (e.g. multiple inheritance). This problem is solved by defining some constraints to restrict the usage of forbidden elements. The resulting model is called restricted (R)-PIM. On the other hand, OPC UA has mandatory node attributes like *'NodeId'* and *'DisplayName'* which are added to the UML diagram using a profile.

The transformation from UML to OPC UA (red circle in Figure 3.1) was done manually at the time this paper was written. The automation of this process is the work of

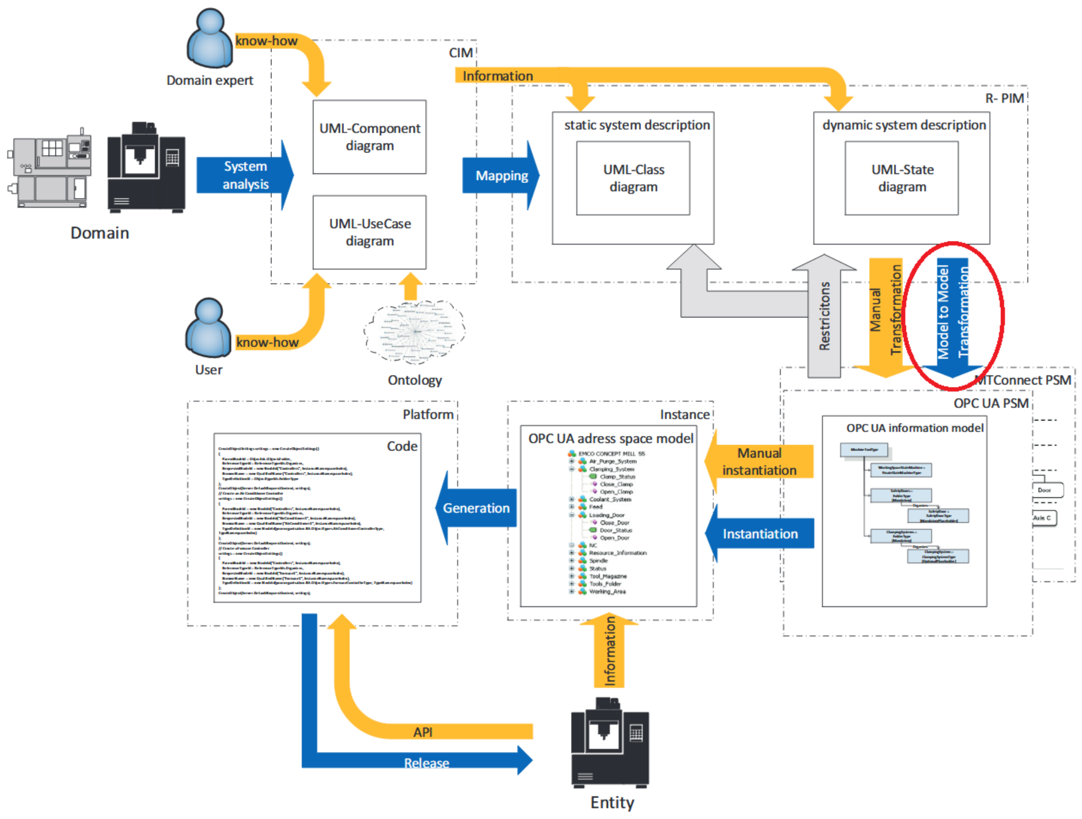


Figure 3.1: MDA based workflow for OPC UA information model design [5]

this thesis. This transformation follows a set of rules which are presented in [1] and discussed in detail in Chapter 4. The result of this PIM to Platform Specific Model (PSM) transformation is an OPC UA information model. Next, the information model is instantiated for a specific entity. This results in an OPC UA address space model (cf. [6]).

3.2 UML2OPC-UA – Transforming UML class diagrams to OPC UA information models

Referring [1]

This paper aims to overcome the big implementation complexity of OPC UA. *Pauker et al.* are trying to do so by enabling an automatic M2M transformation from UML class diagrams to OPC UA information models using ATLAS Transformation Language (ATL) and extending UML to guarantee information preserving transformations. UML class diagrams are used to describe the static bahaviour and UML state machines are used for the dynamic behaviour of a system.

UML Concept	OPC UAConcept	Comment
Class	UAObjectType	mapped
	nodeID	«BasicAttributes».ID
	browseName	«BasicAttributes».BrowseName
	displayName	«BasicAttributes».DisplayName
isAbstract	isAbstract	mapped
superClass	HasSubtype	inverse
ownedOperation	HasComponent	mapped
ownedAttribute	HasProperty	mapped
Enumeration	UADataType	mapped
ownedLiteral DataType-Field	mapped	
Property	UAVariableType	mapped
Operation	UAMethod	mapped
	HasModellingRule	«AdditionalAttributes».ModellingRule
	ParentNodeID	derived (NodeId from the related Class)
Parameter	UAVariable	mapped
	Reference	derived (depends on the different relations in UML)
	UAObject	derived (for Composition Relations in OPC UA supplemented)

Table 3.1: Mapping between UML and OPC UA(excerpt)

The most important transformation mapping rules are summarized in Table 3.1. The "« »" notations define the used package in UML diagrams. They are examples of the additional information needed in a UML class diagram for the mapping.

Furthermore, the paper explains the mapping in greater detail and provides simple examples on how single UML classes, attributes, operations and compositions are realized in OPC UA.

The paper concludes with the idea to implement the presented M2M transformation approach. This idea is the starting point of this thesis.

CHAPTER 4

UML to OPC UA model transformation

This chapter contains the explanation of the MDA approach used in this thesis. The first section contains the explanation of the UML XMI constructs, which are relevant for the M2M transformation, followed by the transformation rules. A transformation example is provided at the end of the chapter.

4.1 UML XMI constructs

UML element	XMI representation	Comment
class	<packagedElement type="uml:Class"/>	
attribute	<ownedAttribute/>	has a parent (e.g. class)
operation	<ownedOperation/>	represents a method, has a parent (e.g. class)
parameter	<ownedParameter/>	child of an operation
primitiveType	<packagedElement type="uml:PrimitiveType"/>	in addition to the predefined types
association	<packagedElement type="uml:Association"/>	is not really needed for the transformation
generalization	<generalization/>	has a parent (e.g. class), supertype in "general" attribute

Table 4.1: Core UML elements and their XMI represenation

Modelio's UML M2T transformation (XMI export) needs to be analyzed in order to enable the transformation between UML and OPC UA. Table 4.1 shows the XMI representation of the most important UML constructs.

Additionally, following important rules apply to these elements:

- Everything is a child of *<uml:Model/>* (root XML node).
- Each XML element has an id attribtue (*xmi:id*) which is unique.
- An element's XMI representation has a *"name"* attribute if the UML diagram defines a name for the element.
- Each *Class* and *PrimitiveType* is represented in a "*packagedElement*" with all its members as children.
- An *ownedAttribute* can either be
  - A *Class* variable or
  - A declaration of an association if the *"association"* and/or *"aggregation"* attributes are present.
- The type of a variable is determined via
  - The id in the *type* attribute which refers to a *PrimitiveType* or
  - the *<type/>* child if the variable's type is one of the predefined types of Modelio.

An association is defined via the *"<ownedAttribute/>"* child element in its parent element where the *"type"* attribute contains the id of the associated element. In this case, the *"<ownedAttribute/>"* element has an "*aggregation*" attribute. For this thesis, the only interesting aggregation is the composition, which means the "*aggregation*" attribute should have the value "*composite*". An association's XML element (*packagedElement* with type "*uml:Association*") is not necessary in order to identify the relations between two elements. The only additional information it offers is the association's parent role label in the *"name"* attribute of its child *"ownedEnd"* which is irrelevant for the transformation.

Listing 4.1 shows an example UML XMI export of a class *"Person"* containing one variable called *"name"*. The *"type"* attribute refers to another id which is not visible here.

The XMI representations of UML models can also contain additional information like *visibility* (e.g. *"name"* is declared *public* in Listing 4.1). Many of these additional attributes have default values and are not mentioned explicitly if they are not changed in the diagram. Table 4.2 lists the most important XML attributes available in the UML M2T transformations.

```
<packagedElement xmi:type="uml:Class"
   xmi:id="_9I6ihX4-EeeatJyGjfHDsg" name="Person">
  <ownedAttribute xmi:id="_9I6ihn4-EeeatJyGjfHDsg" name="name"
     visibility="public" type="_9I6ihH4-EeeatJyGjfHDsg"
     isUnique="false"/>
</packagedElement>
```

Listing 4.1: XMI representation of a class with one attribute.

XMI element	Attributes	Comment
<packagedElement/>	xmi:type	type of element (e.g. *"class"*)
	xmi:id	unique id
	xmi:name	
<ownedAttribute/>	xmi:type	id of type of variable or id of association end
	xmi:id	unique id
	xmi:name	
	optional	
	value	constants or default value
	aggregation	kind of aggregation (e.g. *"composition"*)
	association	contains the id of the *uml:Association*, can be ignored
<defaultValue/>	xmi:type	mostly *uml:LiteralString*
	xmi:id	unique id
	value	
	symbol	value attribute if type is not *uml:LiteralString*
<ownedOperation/>	xmi:id	unique id
	xmi:name	
<ownedParameter/>	xmi:type	type of element (e.g. *"class"*)
	xmi:id	unique id
	xmi:name	
	direction	in/out or return
<generalization/>	xmi:id	unique id, is child of subtype
	general	contains id of supertype
<upperValue/>	xmi:id	unique id, is child of attribute
	value	contains multiplicity information
<lowerValue/>	xmi:id	unique id, is child of attribute
	value	contains multiplicity information

Table 4.2: Attributes of the relevant XML elements

4.2 Transformation rule set

The transformation rule set is based on the paper presented in Section 3.2. This section extends and explains these rules in detail. The transformation rules for classes and compositions, attributes and variables, and operations have their own subsection, respectively. All tranformation rule figures (Figure 4.1, Figure 4.3 and Figure 4.2) follow the same convention: the original UML element is given on the left side, a transformation comment can be found in the middle and the transformation result is on the right side.

4.2.1 Classes and compositions

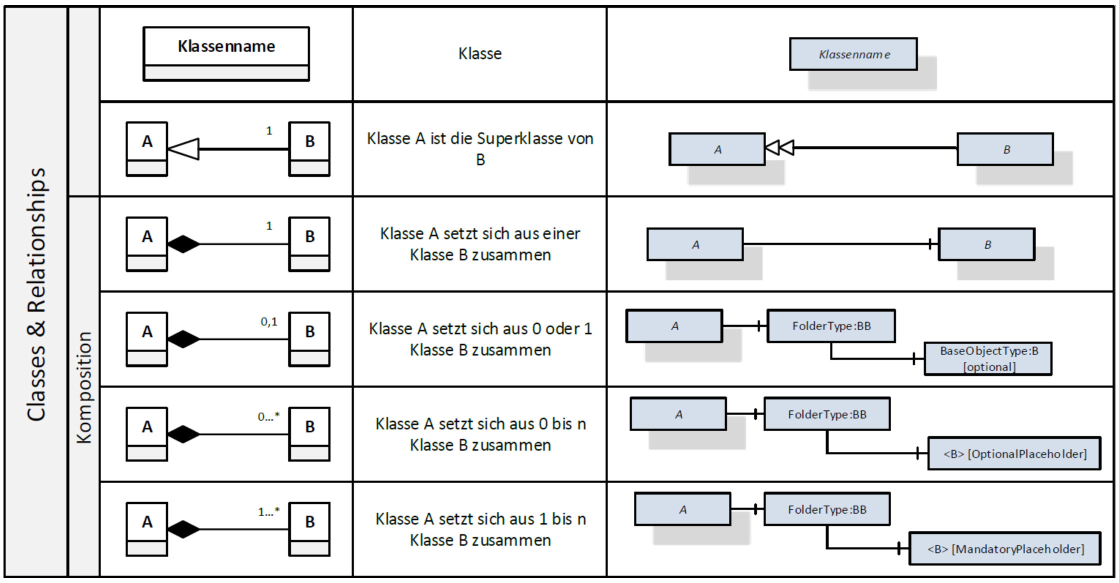

Classes & Relationships
Komposition
Klassenname
Klasse
Klassenname
A
1
B
Klasse A ist die Superklasse von B
A
B
A
1
B
Klasse A setzt sich aus einer Klasse B zusammen
A
B
A
0,1
B
Klasse A setzt sich aus 0 oder 1 Klasse B zusammen
A
FolderType:BB
BaseObjectType:B [optional]
A
0...*
B
Klasse A setzt sich aus 0 bis n Klasse B zusammen
A
FolderType:BB
<B> [OptionalPlaceholder]
A
1...*
B
Klasse A setzt sich aus 1 bis n Klasse B zusammen
A
FolderType:BB
<B> [MandatoryPlaceholder]

Figure 4.1: Transformation of composition reference [1]

Figure 4.1 shows the simplified mapping between UML classes and their respective representation in OPC UA. Basically, UML classes are transformed into *ObjectTypes* in OPC UA. Hierarchical references are significantly different in both modelling languages. In UML, a superclass relation is defined as a generalization of the subtype, thus the link between hierarchical references starts at the subtype. A class in UML does not necessarily "know" it has a subtype. However, OPC UA defines a *"HasSubType"* reference which realizes the same hierarchical reference but the link starts in the supertype. In the graphical notation of OPC UA, the *"HasSubtype"* arrow is then reversed resulting in an arrow pointing in the same direction as the arrow in the original UML diagram.

This thesis' transformation approach realizes compositions in OPC UA as ObjectTypes having a FolderType component which organizes the composing classes. A *FolderType* is a subtype of *BaseObjectType*. The purpose of a folder is to organize other nodes in the address space [4].

Furthermore, this thesis uses a naming convention for transformed nodes. Principally, an OPC UA node keeps the name of its correspondent UML element with following additions:

- *Objects* have *"_Object"* postfixed.
- *ObjectTypes* have *"Type"* postfixed.
- *FolderTypes* have *"_Folder"* postfixed.
- *Placeholders* have their name within *"<>"* brackets.

4.2.2 Operations

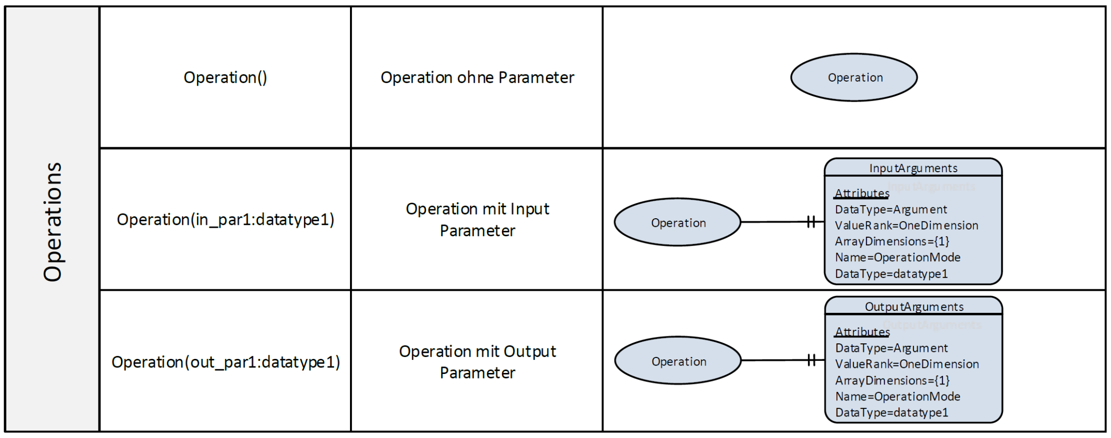


Figure 4.2: Transformation of operations [1]

Figure 4.2 shows the transformation rules for operations. In OPC UA, operations have their own node type: *Method*. *Methods* have *HasProperty* references to their input- and output arguments. These are realized as *Variables* with the *DataType* "Argument" as their properties. Arguments can either be *InputArguments* or *OutputArguments*. The number of arguments is specified in the *ArrayDimensions* property. Details about arguments are specified in the *Value* attribute of the *VariableType* and are called a "List of Extension Objects". The return value of an operation is a value member of *OutputArgument* with the name *"ReturnValue"*.

4.2.3 Attributes and variables

Figure 4.3 shows the transformation rules for variables. In principle, UML variables simply become a *VariableType* in OPC UA. The proposed rule set offers a special name syntax to declare variables static (s), dynamic (d) or optional (o) which is specified in round brackets. It is also possible to specify units for variables. The unit's name has to be put in square brackets. Additionally, it is possible to declare whether the variable is

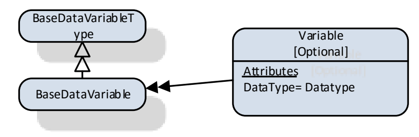
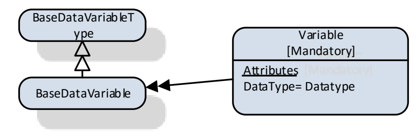
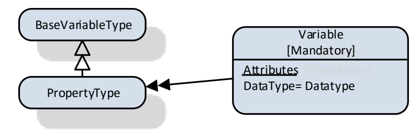
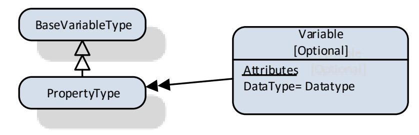
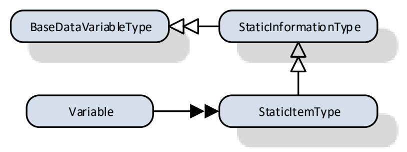
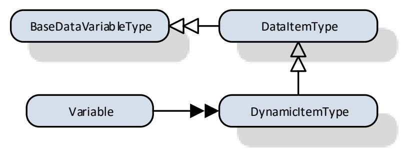
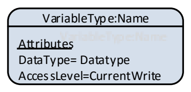
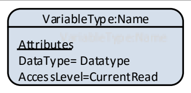
Attributes & Variables	TypeDefinition	-Variable(od):Datatype	Statisches optionales Attribut ohne Einheit	 BaseDataVariableType BaseDataVariable Variable [Optional] Attributes DataType= Datatype	
		-Variable(d):Datatype	Veränderliches Attribut ohne Einheit	 BaseDataVariableType BaseDataVariable Variable [Mandatory] Attributes DataType= Datatype	
		-Variable(s):Datatype	Attribut ohne Einheit	 BaseVariableType PropertyType Variable [Mandatory] Attributes DataType= Datatype	
		-Variable(os):Datatype	Optionales Attribut ohne Einheit	 BaseVariableType PropertyType Variable [Optional] Attributes DataType= Datatype	
		-Variable[Unit](s):datatype	Attribut mit Einheit, der Wert ändert sich nicht	 BaseDataVariableType StaticInformationType Variable StaticItemType	
		-Variable[Unit](d):datatype	Attribut mit Einheit, der Wert ändert sich dynamisch	 BaseDataVariableType DataItemType Variable DynamicItemType	
	UserAccess	-Variable:datatype{w}	Attribut mit Schreibberechtigung	Attribut AccessLevel = CurrentWrite	 VariableType:Name Attributes DataType= Datatype AccessLevel=CurrentWrite
		-Variable:datatype{r}	Attribut mit Leseberechtigung	Attribut AccessLevel = CurrentRead	 VariableType:Name Attributes DataType= Datatype AccessLevel=CurrentRead

Figure 4.3: Transformation of attributes and variables [1]

readOnly or writeOnly via the letters "r" and "w" written in curly brackets. The default attributes of variables are: dynamic, read/write and no unit.

In OPC UA optional, mandatory and read-/write access properties can be specified directly as *VariableType* properties. To specify static variables (constants) the *VariableType* is declared as a *PropertyType* instead of a *BaseDataVariableType*. This thesis' units realizations differ from the reference approach. Here, units are realized as *VariableType* nodes with the unit's name as the node's *DisplayName*. Other variables can have a *HasProperty* reference on a unit node. This approach can be found in the OPC UA specification [4].

4.3 The transformation in use

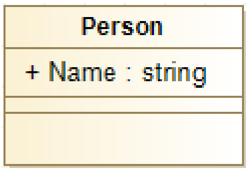


Figure 4.4: UML diagram of a *Person* with a *Name*

The last section of this chapter presents a simple transformation example. Figure 4.4 shows the UML class *Person* with one argument *name*. Its XMI representation can be seen in Listing 4.1.

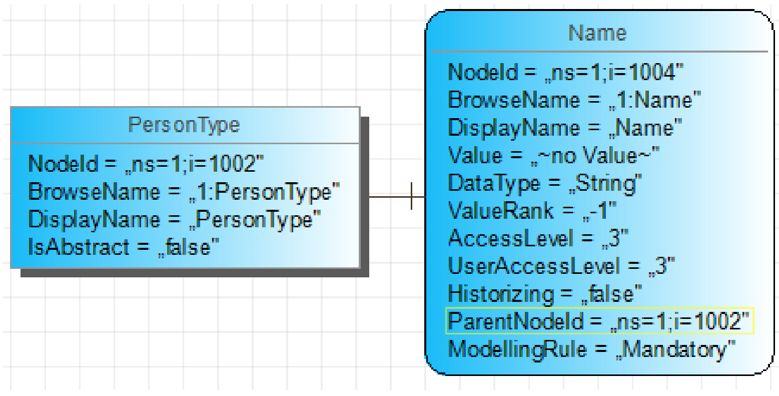


Figure 4.5: OPC UA information model created by a transformation

Figure 4.5 shows the product of a finished UML to OPC UA transformation. As proposed in the previous section, the class *Person* becomes an *ObjectType* node in OPC UA and its attribute *name* becomes a *Variable* node. *Name* is a property of *Person* which is shown with the *HasProperty* reference between both nodes.

CHAPTER 5

Proof of Concept

This chapter presents the realization of the motivation of this thesis. After explaining the relevant modelling languages in Chapter 2 and explaining the modelling rules in the previous Chapter 4, the only thing left to do is choosing an open source modelling tool, contribute to it and put the presented model transformation to use. This thesis uses an open source UML modelling tool as a starting point rather than an OPC UA one because UML has more advanced tools and much more support. This is why this chapter starts with presenting and comparing UML modelling tools before presenting the usage of the UML to OPC UA model transformation in the most useful and therefore modified open source UML modelling tool.

5.1 Open source UML modelling tools

Table 5.1 shows a list of open source UML modelling tools which were considered for the implementation of the presented model transformation approach. Eclipse UML2 Tools and Papyrus are both available as a plugin for Eclipse [7]. These two programs were ruled out immediately because while testing both programs kept crashing with even the most basic inputs. Using these programs seemed infeasible and they are not further discussed.

Name	**Latest stable release**	**Programming language**
ArgoUML	15.12.2011	Java, C++
Eclipse UML2 Tools	19.02.2009	Java
Papyrus	27.06.2013	Java
Modelio	02.10.2018	Java
EasyUML (NetBeans)	21.02.2013	Java

Table 5.1: List of open source UML modelling tools

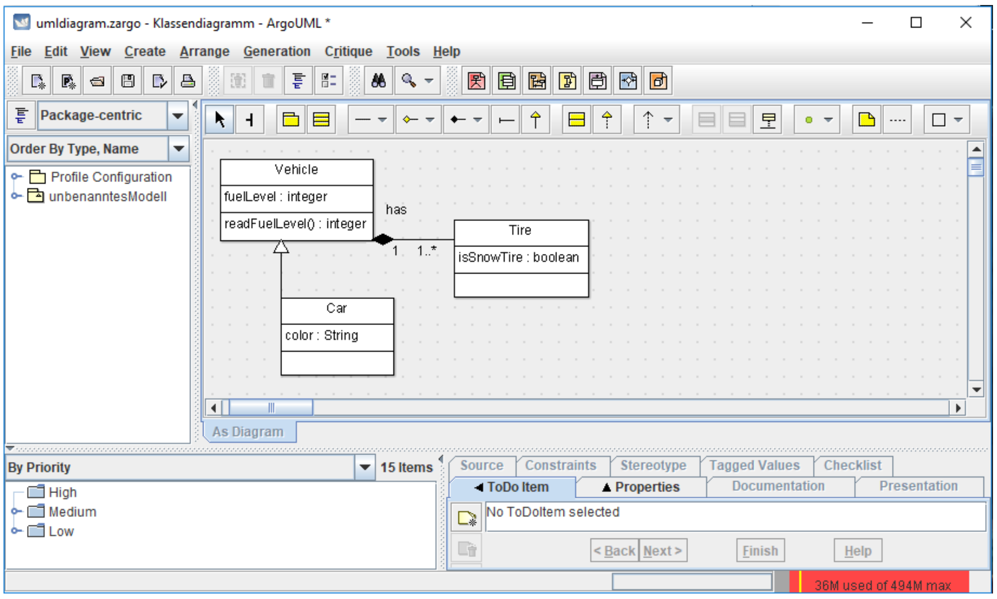

umldiagram.zargo - Klassendiagramm - ArgoUML *
File Edit View Create Arrange Generation Critique Tools Help
Package-centric
Order By Type, Name
Profile Configuration
unbenanntesModell
Vehicle
fuelLevel : integer
readFuelLevel() : integer
has
1
1..*
Tire
isSnowTire : boolean
Car
color : String
As Diagram
By Priority
15 Items
High
Medium
Low
Source Constraints Stereotype Tagged Values Checklist
ToDo Item Properties Documentation Presentation
No ToDoItem selected
< Back Next > Finish Help
36M used of 494M max

Figure 5.1: ArgoUML GUI with a diagram

Figure 5.1 depicts an example diagram created with ArgoUML [8]. ArgoUML claims to be the leading open source UML modelling tool and supports all standard UML 1.4 diagrams. The program is straight forward, easy to use with a simple user interface. It is possible to export and import UML diagrams as XMI file. However, it does not use the OMG UML XMI 2.1 schema from [9]. It appears to use its own export convention.

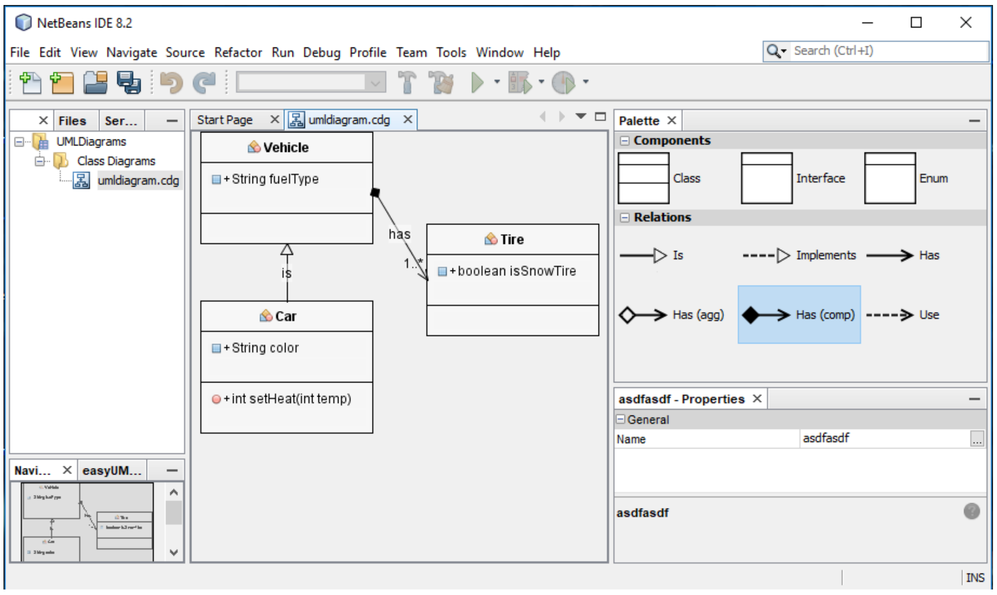
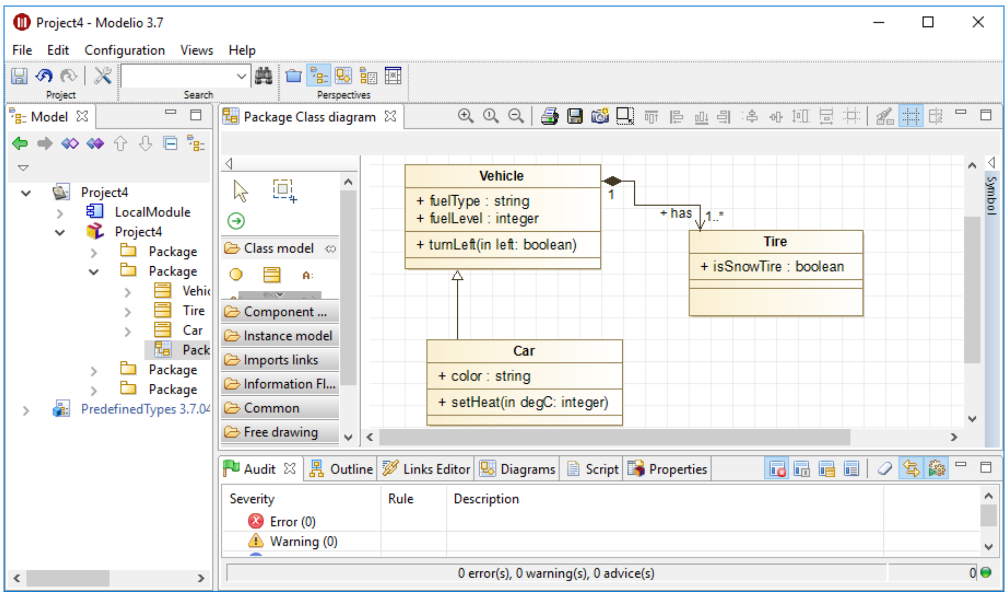


Figure 5.2: EasyUML (NetBeans) GUI with a diagram

Figure 5.2 shows an example diagram created with EasyUML, a plugin from NetBeans [10]. EasyUML has an even simpler user interface. It only supports the most basic UML elements. Without additional plugins it is neither possible to export nor import diagrams in any way. It is, however, possible to create Java code from diagrams.



0 error(s), 0 warning(s), 0 advice(s)

Figure 5.3: Modelio GUI with a diagram

Figure 5.3 shows an example diagram created with Modelio 3.7 [11]. This program has an easy user interface. It supports UML2 and the standardized XMI import and export of UML diagrams defined by the OMG [9]. This modelling tool is often updated and has an active community forum. Modelio is an Eclipse Rich Client Platform (RCP) application which allows Modelio to be designed component based.

Modelio's technical superiority in comparison to the previously mentioned UML modelling tools and its modular code base makes it a clear choice to work with in this thesis. It is possible to add the UML transformation and the ability to design OPC UA information models without changing much of the existing code.

5.2 Modelio with OPC UA extension

Figure 5.4 shows a version of the work done for this thesis: Modelio 3.7 with the OPC UA extension. It can be seen that with the extension it is possible to design OPC UA information models. The extension basically adds an additional diagram type (*OPCUADiagram*), all node types and the most important reference types which are described in Figure 2.4.

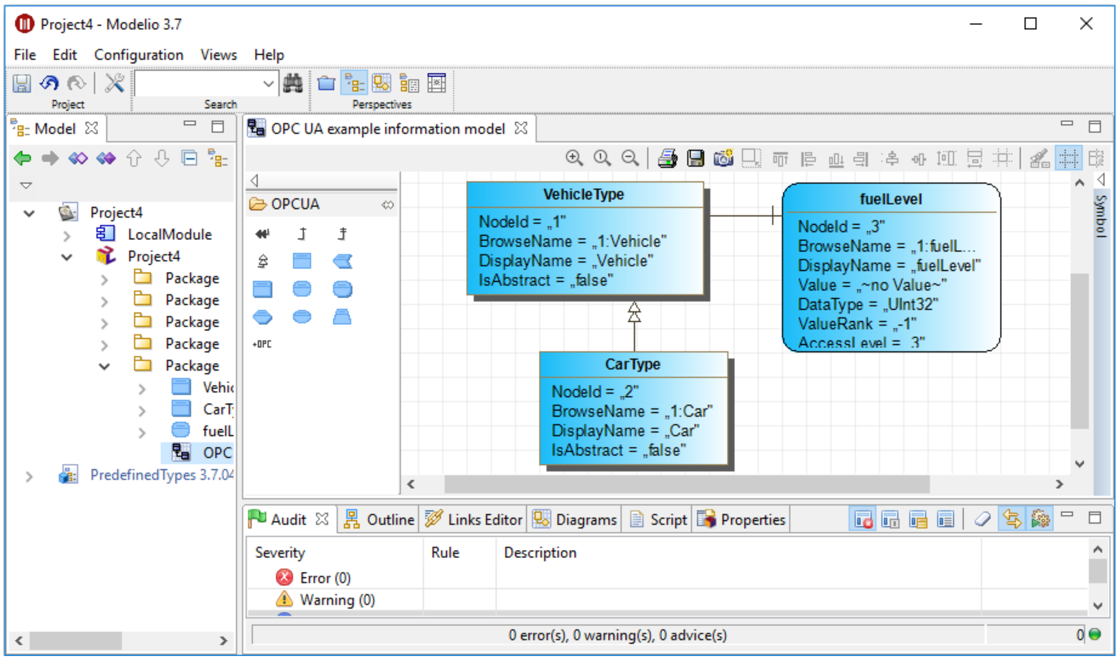


Figure 5.4: Modelio 3.7 GUI with OPC UA extension

5.2.1 Graphical user interface

The Graphical User Interface (GUI) for OPC UA information models works similar to the UML diagrams. To create an OPC UA information model in Modelio the user has to create a diagram (right click on a Package in the project tree view) and choose the *OPCUA Diagram* in the Creation Wizard. This option opens an empty diagram just like the class diagram, but with a new palette (clickable icons just left of the information model: see Figure 5.4). This is a feature provided by Eclipse RCP and is added similarly to existing palettes. The palette contains all the node types and the most important reference links from Figure 2.4 with their respective standardized shapes.

It is possible to drag-and-drop nodes from the palette into the diagram. The created node automatically gets its mandatory attributes according to the OPC UA specification [4]. To create a specific reference link between two nodes a link icon has to be clicked and then both of the nodes in question need to be selected in the from-to order. Note that the *HasSubType* reference is reversed accordingly. Additionally, it is possible to give nodes extra attributes by clicking the *"+OPC"* icon and then the node in question. This can be used to add one of the optional attributes or new attributes unfamiliar to the OPC UA standard. The latter is ignored in the transformation process. This can be seen as a potentially useful modelling feature.

Furthermore, it is possible to import and export the information models by right clicking the Package in which the model was created and choose *Import/Export* and then *OPCUA Import* to import valid OPC UA nodeset files, which are XML files. Additionally, it is possible to transform and export the information model into its textual representation by clicking *OPCUA to XML export*. The imported and exported nodeset files are checked for validity with the XML schema from [12].

5.2.2 Back-end

OPC UA nodes

In the back-end, the nodes are all a subtype of *UANode* which is a subtype of *Classifier* (see Figure 2.1). This is done purely out of simplicity because the supertype *Classifier* provides every *UANode* with all the modelling abilities they need. *UANode* is basically a copy of *Class* with different properties. For instance, *UANodes* have to get rid of the groupings of attributes, operations and inner classes (see the three rows after a classe's name in Figure 4.4) since they can only have *OPCUAAttributes* (e.g. *"NodeId"*,...).

OPC UA references

A similar technique is used for the references which are all a subtype of *Generalization* because this provides them with the ability to link two nodes without additional coding. After that, it is possible to define every subtype of *UANode* as its own diagram element which provides them with the ability to define their own figure. Similarly, references get their respective shapes.

Afterwards, modelling rules need to be defined. Link rules are defined in *DefaultLinkExpert.java* and structural rules are stated in *DefaultMetaExpert.java*. Listing 5.1 shows an example of a link and structural rules. The provided example would allow the placement of *OPCUAAttributes* inside *OPCUAObjects* and a *HasSubType* reference between two *OPCUAObjectType* nodes.

```
addRule(OPCUAObject.MQNAME, OPCUAAttribute.MQNAME);
---
addRule(HasSubType.MQNAME, OPCUAObjectType.MQNAME,
    OPCUAObjectType.MQNAME);
```

Listing 5.1: UA modelling rules

5.3 Model transformation in Modelio

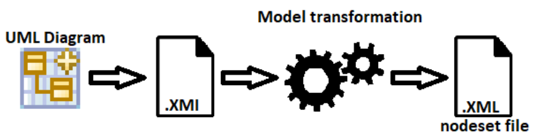


Figure 5.5: Illustration of the model transformation process

The transformation process has been implemented as an independent program which takes a UML diagram in its XMI representation as input and outputs an OPC UA nodeset file. It builds upon the tranformation rules described in Section 4. Figure 5.5 shows the simplified procedure of the model transformation. To make use of the proposed M2M UML to OPC UA transformer program in Modelio, a user has to first create a UML diagram of a system which then needs to be exported. To do so, the *Package* in which the diagram was created needs to be right clicked and *Import/Export* → *OPCUA Export* needs to be chosen. After declaring a save directory and a file name for the transformed model, the transformation starts.

5.3.1 Model transformation algorithm

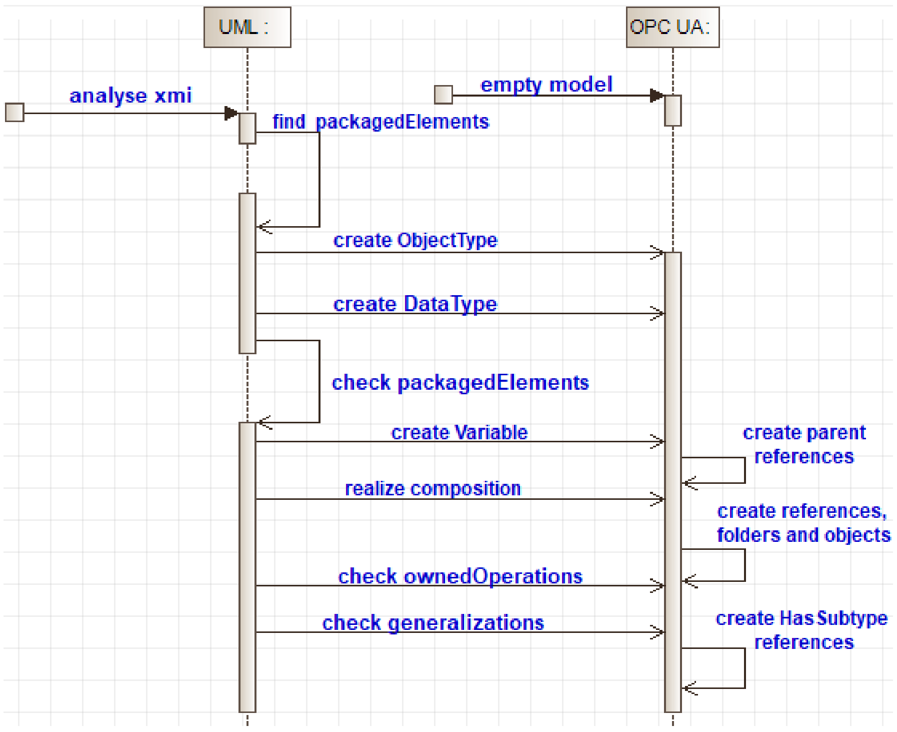


Figure 5.6: Schematic workflow of the UML to OPC UA transformation

The transformation program is an XML parser which uses the Java DOM Parser (*javax.xml.parsers.** and *org.w3c.** packages) for reading the input file and creating the nodeset file. The workflow of the program can be seen in the sequence diagram in Figure 5.6. The input file is first parsed for relevant *packagedElemet*s (see Table 4.1). The found *class*es and *primitiveType*s are then mapped to their OPC UA counterparts and saved in their nodeset file representation. The nodes are given an incrementing id starting from 1002. This starting number has been chosen because it is possible to assign several thousand ids without interfering with any predefined indices. Their *xmi:id* as well as their newly given OPC UA id in combination with their respective *DOM Element* are stored in *HashMap*s for future references. This is why *packagedElement*s are all resolved before checking their children. A child can be a hierarchical reference to a succeeding element.

Sequentially, all found *packagedElement*s are parsed for each of their children. Firstly, every *ownedAttribute* is checked for its functionality. Therefore, variables become OPC UA variables with either a *HasProperty* or *HasComponent* reference to their parents depending on its definition (see Chapter 4). If the *ownedAttribute* turns out to specify a composition, it is realized accordingly in the nodeset file. Afterwards, the *packagedElement* is checked for *ownedOperation*s. These are simply mapped to their OPC UA counterpart. Lastly, generalizations are realized as *HasSubtype* references in the OPC UA information model just as proposed. The finished OPC UA information model node set file is checked for validity with its XML Schema Definition (XSD) from [12].

Example transformation result

```
<UANodeSet [...]>
  <NamespaceUris>
    <Uri>http://www.umltoopcua.com/yourproject</Uri>
  </NamespaceUris>
  <Aliases>
    <Alias Alias="String">i=12</Alias>
    [...]
  </Aliases>
  [...]
  <UAObjectType BrowseName="1:PersonType" NodeId="ns=1;i=1002">
    <DisplayName>PersonType</DisplayName>
    <References>
      <Reference IsForward="false"
          ReferenceType="HasSubtype">i=58</Reference>
      <Reference ReferenceType="HasComponent">ns=1;i=1004</Reference>
    </References>
  </UAObjectType>
  <UAVariable AccessLevel="3" BrowseName="1:Name" DataType="String"
      NodeId="ns=1;i=1004" ParentNodeId="ns=1;i=1002"
      UserAccessLevel="3">
    <DisplayName>Name</DisplayName>
    <References>
      <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
      <Reference ReferenceType="HasModellingRule">i=78</Reference>
      <Reference IsForward="false"
          ReferenceType="HasComponent">ns=1;i=1002</Reference>
    </References>
  </UAVariable>
</UANodeSet>
```

Listing 5.2: Nodeset file created by transforming Listing 4.1.

Listing 5.2 shows a transformation example. The input of the program was Listing 4.1. The class *Person* becomes an *ObjectType* and its variable *Name* becomes a *VariableType* in OPC UA. There is a *HasComponent* link from *PersonType* to *Name* to showcase their hierarchical reference. Mandatory references to the node's type definitions and OPC UA base types are made automatically. The output files are edited so they can be further worked on and imported by UaModeler [6]. Predefined indexes (e.g. i=12 for the *DataType* "*String*") were observed and are used likewise in the transformer.

5.3.2 Adding the model transformation algorithm to modelio

All **.java* files needed by the transformation are added as a new package in Modelio's *"xmi"* project folder. The transformation is added similarly to the existing imports and exports of UML diagrams. A new window design is defined which fits accordingly to the needs of all OPC UA import and export informations (e.g. textbox for file path). Additionally, a new thread for all three newly added operations (import, and export of OPC UA information models and the export of UML to OPC UA transformation) is created which uses the respectively needed transformation objects.

CHAPTER 6

Conclusion and future work

Motivated by the lack of OPC UA modelling tools, especially on the open-source market, this thesis presented Modelio (an open-source UML modelling tool) with an OPC UA information model extension. This extension enables graphical modelling of OPC UA information models in Modelio. Furthermore, these models can be exported as nodeset files by making use of M2T transformations.

This thesis also presented a slightly modified approach from [1] for creating OPC UA information models by transforming UML diagrams. This process was used to bypass the modelling complexity of OPC UA models. The approach consisted of a mapping between UML and OPC UA elements. UML was constrained by additional OPC UA friendly rules to ensure UML could not use any features which were not supported by OPC UA. The transformation approach was implemented in *Java* as an independent program. It used the *Java DOM Parser* to parse the standardized XML representation (XMI) of UML diagrams and for constructing the textual representation of OPC UA nodeset files.

Modelio was chosen as a starting point because it was considered the most suitable and best open-source UML modelling tool available. Open-source OPC UA modelling tools seemed infeasible as a starting point for this thesis because they were technically by far inferior compared to their UML counterparts. Modelio was given a new OPC UA diagram type and a new GUI for modelling OPC UA information models and the M2M UML to OPC UA transformation program were also added.

The transformation worked pretty well in Modelio. In a next step, Modelio's OPC UA modelling capabilities should be improved, since Modelio neither checks for valid OPC UA models nor respects any special OPC UA rules. It would also be wise to let Modelio and the model-UA repository [13] stay up to date, especially when future UML to OPC UA mapping are released since transformation rules for UML state charts are currently worked on [1].

List of Figures

List of Tables

Acronyms

ATL ATLAS Transformation Language. 12

CIM Computation Independent Model. 6, 11

CPPS Cyber-Physical Production System. ix, 1

CPS Cyber-Physical System. ix, 1

GUI Graphical User Interface. 26, 31

M2M Model-to-Model. 2, 6, 9, 12, 13, 15, 28, 31

M2T Model-to-Text. 9, 16, 31

MDA Model-Driven Architecture. 9, 11, 15

MDE Model-Driven Engineering. 9

OMG Object Management Group. 5, 9, 24, 25

OPC UA OPC Unified Architecture. vii, ix, xii, 1–3, 5–8, 11–13, 16, 18–21, 23, 25–31, 33, 34

PIM Platform Independent Model. 5, 6, 11, 12

PSM Platform Specific Model. 12

RCP Rich Client Platform. 25, 26

TCP Transmission Control Protocol. 6

UML Unified Modelling Language. vii, ix, 1, 2, 5–8, 11–13, 15, 16, 18, 19, 21, 23–26, 28, 30, 31, 33, 34

XMI XML Metadata Interchange. 2, 11, 15–17, 21, 24, 25, 28, 31, 34

XML Extensible Markup Language. 2, 3, 6, 9, 16, 17, 26–29, 31, 34, 35

XSD XML Schema Definition. 29

Bibliography

[1] Florian Pauker, Sabine Wolny, Solmaz Mansour Fallah, and Manuel Wimmer. UML2OPC-UATransforming UML Class Diagrams to OPC UA Information Models. *Procedia CIRP*, 67:128–133, 2018.

[2] OMG UML specification. `"https://www.omg.org/spec/UML/2.5.1/PDF"`. Accessed: 04-10-2018.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The 2nd Edition; ISBN-13: 978-0-321-26797-9.*

[4] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture.* ISBN: 978-3-540-68898-3.

[5] Florian Pauker, Thomas Frühwirth, Burkhard Kittl, and Wolfgang Kastner. A systematic approach to OPC UA information model design. *Procedia CIRP*, 57:321–326, 2016.

[6] Unified Automation - UA Modeler. `"https://www.unified-automation.com/products/development-tools/uamodeler.html"`. Accessed: 04-10-2018.

[7] Eclipse. `"https://www.eclipse.org/"`. Accessed: 04-10-2018.

[8] ArgoUML. `"http://argouml.tigris.org/"`. Accessed: 04-10-2018.

[9] OMG UML XMI standard. `"http://schema.omg.org/spec/XMI/2.1"`. Accessed: 04-10-2018.

[10] NetBeans. `"https://netbeans.org/"`. Accessed: 04-10-2018.

[11] Modelio. `"https://www.modelio.org/"`. Accessed: 04-10-2018.

[12] OPC Foundation nodesetfile schema 1.03. `"https://opcfoundation.org/UA/schemas/1.03/UANodeSet.xsd"`. Accessed: 04-10-2018.

[13] model-UA: UML to OPC UA model transformation. `"https://github.com/model-UA/uml-to-opcua"`. Accessed: 04-10-2018.