FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Adaptable OCL Engine for Validating Models in Different Tool Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-IngenieurIn

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Christoph Zehetner
Matrikelnummer 0626098

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Mag. Dr. Manuel Wimmer, Dr. Horst Kargl

Wien, 01.07.2013

(Unterschrift VerfasserIn) (Unterschrift BetreuerIn)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Adaptable OCL Engine for Validating Models in Different Tool Environments

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-IngenieurIn

in

Business Informatics

by

Christoph Zehetner
Registration Number 0626098

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Mag. Dr. Manuel Wimmer, Dr. Horst Kargl

Wien, 01.07.2013

(Signature of Author) (Signature of Advisor)

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

Erklärung zur Verfassung der Arbeit

Christoph Zehetner, Kirchenstraße 9, 2225 Loidesthal

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.

Abstract

The software engineering process has significantly changed over the last decade. The paradigm Model-Driven Development (MDD) gained more and more popularity. In the past, models were typically used for brainstorming and communication purposes. This viewpoint has shifted dramatically. The software engineering process becomes more model-centric and less code-centric. Thus, models are the key artefacts in the development process and all steps rely on these models and their correctness. Sophisticated modelling techniques have been invented to ensure a consistent and comprehensive technology base. The standard for modelling structural and behavioural aspects of systems, the Unified Modelling Language (UML) was introduced by the Object Management Group (OMG). The problem is that an UML model is not necessary expressive enough to provide all complex aspects of a system. Some aspects of the domain are more expressed in natural language. Practise has shown that such situations will lead to ambiguities and to errors. Therefore the Object Constraint Language (OCL) can be used to specify conditions on models that must hold for the system.

Apart from the development of standards for modelling systems, several vendors developed Computer-Aided Software Engineering (CASE) tools to provide a wide range of instruments to support the MDD approach. The support of the sophisticated techniques depends on the vendor's realization of the standards, which is different from tool to tool. In general there is a lack of validating models in CASE tools, which is essential in the early design phase. Additionally, there is only little support for writing OCL expressions for models. Furthermore, models defined by the users of the tool are not validated against the well-formedness rules (WFR) - described with OCL - of the UML specification. An automatic validation of the OCL expressions of the UML specification before code generation allows the detection of errors in the early design phase and reduces problems for the further progress of projects. Thus, it is a huge value that CASE tools support the definition and validation of OCL expressions.

The contribution of this master's thesis is to provide an adaptive solution for managing OCL transformation in different environments or CASE tools, called ADOCLE. Therefore mapping patterns are defined to manage the schema matching between selectable environments (source and target schema). The OCL expressions are applied to a selectable source schema, which is mapped to a target schema. The goal of this work is to analyse OCL expressions and generate semantically equivalent expressions in the target schema depending on predefined transformation patterns.

Kurzfassung

Die Computerwissenschaft beschäftigt sich schon seit vielen Jahren mit Modellierungstechniken zur Beschreibung von Strukturen, Architekturen und Prozessen in Softwareprojekten. Anfänglich wurden Modelle für reine Kommunikations- und Entwurfszwecke eingesetzt. Im Laufe der letzten Jahre gewann die modellgetriebene Softwareentwicklung, auch bekannt als *Model-Driven Development (MDD)*, immer mehr an Bedeutung, womit die Modelle zum Kernstück des Entwicklungsprozesses wurden. Basierend auf den Modellen können Teile des Systems (Source Code) generiert und weiter verarbeitet werden. Dabei spielt vor allem die Gültigkeit der Modelle eine tragende Rolle. Verschiedenste Modellierungstechniken wurden entwickelt, um eine fundierte Basis für MDE zu schaffen. Die *Object Management Group (OMG)* führte daraufhin, einen de facto Standard für die Modellierung von Systemen, die *Unified Modelling Language (UML)* ein. Das Hauptproblem ist, dass ein UML Modell nicht notwendigerweise ausreichend ausdruckstark ist, um alle komplexen Aspekte eines Systems abzubilden. Einige Aspekte werden dadurch in natürlicher Sprache formuliert. Die Praxis hat gezeigt, dass solche Situationen zu Unklarheiten und Fehlern führen. Eine Möglichkeit stellt die *Object Constraint Language (OCL)* bereit, indem Bedingungen für Modelle definiert werden können, welche ebenfalls in den Entwicklungsprozess integriert werden können.

Neben der Entwicklung von Standards für die Modellierung von Systemen, entwickelten mehrere Anbieter, *Computer-Aided Software Engineering (CASE)* Tools, um eine breite Palette von Instrumenten des MDD Ansatzes zu unterstützen. Der Support hängt von der Implementierung der Standards des Anbieters ab und ist somit von CASE-Tool zu CASE-Tool unterschiedlich. In Allgemeinen, ist nur eine mangelhafte Validierung der erstellten Modelle in CASE-Tools möglich, die eine essentielle Rolle in der frühen Entwurfsphase spielen. Darüber hinaus gibt es nur wenig Unterstützung für das Definieren von OCL Ausdrücken für Modelle. Außerdem können Modelle von den Benutzern definiert werden, die nicht den wohlgeformten Regeln in der UML-Spezifikation entsprechen. Eine automatische Validierung der OCL Ausdrücke der UML-Spezifikation vor der Codegenerierung ermöglicht das Entdecken von Fehlern in der frühen Entwurfsphase. Daher stellt es einen großen Mehrwert dar, CASE-Tools, um die Definition und Validierung von OCL Ausdrücken zu erweitern.

Die Arbeit beschäftigt sich mit einem adaptiven Ansatz zur Transformation von OCL Ausdrücken in verschiedene Umgebungen oder CASE-Tools. Dazu werden Mappingmuster definiert, um ein *Schema Matching* zwischen wählbaren Umgebungen (Quell-und Ziel-Schema) zu bewerkstelligen. Die OCL Ausdrücke sind einem wählbaren Quell-Schema zugeordnet, welches auf das Ziel-Schema gemappt ist. Das Ziel dieser Arbeit ist es, den OCL-Ausdruck zu analysieren und einen semantisch äquivalenten Ausdruck für das Ziel-Schema je zu generieren basierend auf vordefinierten Transformationsmustern.

Table of Contents

1 Introduction 1
1.1 Motivation 1
1.2 Problem statement 2
1.3 Result 4
1.4 Methodical approach 5
1.5 Structure of the work 6

2 Introduction of validating UML models using OCL expressions 7
2.1 Unified Modelling Language in a nutshell 7
2.1.1 Metamodel Layering 9
2.1.2 Notation 11
2.1.3 Advantages and disadvantages 14
2.1.4 Motivation example 15
2.2 Object Constraint Language in a nutshell 17
2.2.1 Scope of application of OCL 17
2.2.2 Definition of constraint 18
2.2.3 OCL core concepts 19
2.2.4 Types 22
2.2.5 Expression 25
2.2.6 History and related languages 28
2.3 Summary 29

3 On the Analogy of OCL and SQL 30
3.1 Overview of SQL and RDMS 30
3.1.1 Querying relational database using SQL 31
3.2 Analogy between OCL invariants and SQL queries 33
3.2.1 Exemplary mapping of UML model to relational database 34
3.2.2 Exemplary data for the UML model and the relational database 35
3.2.3 Exemplary mapping from OCL to SQL 37
3.3 OCL2SQL transformation approaches 39

4 Realizing ADOCLE 43
4.1 Overview of the ADOCLE 43
4.2 Architecture of the ADOCLE 45
4.2.1 Metamodel loader 45
4.2.2 OCL Interpreter 53
4.2.3 OCL Transformator 55
4.2.4 OCL Validator 70
4.3 User interface 71

4.4 Development issues ... 72
5 Evaluation ... 73
5.1 Goal ... 73
5.2 Problem and hypothesis ... 73
5.3 Test environment ... 74
5.4 Consistency analysis ... 75
5.4.1 Test data ... 76
5.4.2 Test cases ... 82
5.4.3 Test results ... 97
5.5 Performance analysis ... 97
5.5.1 Test results ... 97
6 Related Work ... 99
6.1 Overview of metamodel-based OCL compiler ... 99
6.1.1 Implementing OCL for multiple Metamodels ... 99
6.1.2 OCL Module in VTMS ... 100
6.1.3 Redesign of the Dresden OCL Compiler ... 101
6.1.4 Summary ... 102
6.2 Tool-support ... 103
6.2.1 Dresden OCL ... 103
6.2.2 Eclipse OCL ... 104
6.3 Validation/Verification ... 106
6.3.1 UML-based Specification Environment (USE) ... 106
6.3.2 HOL-OCL ... 108
7 Conclusion and Future Work ... 109
7.1 Conclusion ... 109
7.2 Future Work ... 111
8 List of figures ... 112
9 List of tables ... 113
10 List of Mapping Examples ... 114
11 References ... 115

1 Introduction

1.1 Motivation

In times of Model Driven Engineering (MDE), the Unified Modelling Language (UML) has been widely accepted as a standard for modelling object-oriented systems. UML is a visual language for specifying, constructing and documenting mainly artifacts of software systems, but it can be also used to describe other systems. UML is developed by the Object Management Group (OMG) [**1**], [**2**]. UML models are used to define structural or behavioural aspects of software systems, starting from requirements analysis to the implementation and maintenance phase. In the early days of UML, models extend more or less the development process for a more understandable documentation. In many cases, the models were defined as an instrument for all included stakeholders in the analysis stage of the project or just created when the implementation was already finished as post-documentation.

As the development paradigm Model-Driven Development (MDD) arises, the software development process becomes more model-centric and less code-centric. Models are no longer exclusively for the documentation of design decisions but rather included as first class components in the development process. The OMG's Model Driven Architecture[1] (MDA) is a conceptual MDD approach (see Figure 1) for vendor-neutral specifications of business and application logic to improve interoperability and portability of the specified model. The challenge is to define a platform-independent model (PIM) of the system that describes the business model without technological boundaries. The transformation from a platform-independent model to a platform-specific model (PSM) adds the technologic aspects of the platform, so with one transformation step it is possible to generate source code automatically out of the platform independent business model.

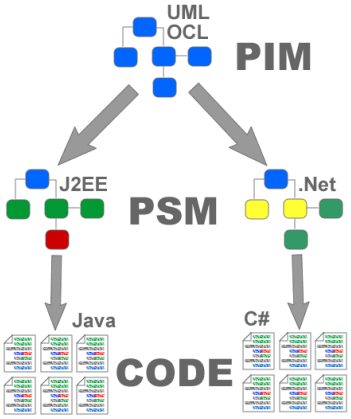

Figure 1 - MDA approach

[1] MDA Definition - http://www.omg.org/mda/

The OMG suggests the usage of UML to specify platform-independent models. A UML model is not necessary expressive enough to provide all relevant aspects of a specification of a software system. Some aspects of the domain are more expressed in natural language. Practise has shown that such situations will lead to ambiguities and to errors. Therefore the Object Constraint Language (OCL) [**3**] can be used to define more formal, but programming language independent expressions. OCL is a formal language to specify invariant conditions on models that must hold for the system. OCL expressions can be integrated in the source code generation during the transformation process to the platform-specific model and validates the entities against the specified model.

With the emergence of agile software development models become more and more important. In every iterative steps of agile methods the requirements and of course the models has to be updated to avoid errors for the next step. As the first law, as Barry Boehm says [**4**], "Errors are most frequent during the requirements and design activities and are the more expensive the later they are removed." it is a significant point of software development to detect errors in the early stage of the development process. OCL expressions are a valuable instrument for managing a validation after each iterative step.

Due to the acceptance of the UML as a standard for modelling software systems, a huge demand for tools supporting an MDD approach arise, especially for case studies, where models need detailed refinements by platform independent expressions. But cover these tools all needs of the developers?

1.2 Problem statement

A lot of Computer-aided software engineering (CASE) tools provide a wide range of instruments to support an MDD approach: drawing and documentation of UML diagrams, automatic code generation and reverse engineering for supporting the developers in the design phase. This support of the UML techniques depends on the implementation of the UML metamodel, which is different from CASE tool to CASE tool.

Some CASE tools prefer a very strictly solution for defining and drawing UML diagram. That means only UML models can be created, which are valid against the implemented UML metamodel (see Poseidon for UML[2] or ArgoUML[3]). The disadvantage of such tools is a long learning curve. Other applications like the Enterprise Architect (EA) [**5**] rely on a more open modelling approach that encourages developers in the creative stage of the design phase. It allows the developers to define models that are based on the UML metamodel, but the models are not 100% UML compliant. Normally, these models are valid against the UML metamodel, but there is no guarantee for it. From structural to behavioural

[2] http://www.gentleware.com/new-poseidon-for-uml-8-0.html

[3] http://argouml.tigris.org/

aspects of software systems most CASE tools provide UML support throughout the development process. But only few applications like “Eclipse OCL” [**6**] facilitate defining OCL expressions and validating object-oriented models. In the case of Eclipse[4], the CASE tools are part of the EMF[5] project, which are part of the integrated development environment (IDE).

However, in general there is a lack of validating models in CASE tools. Particularly, in the design phase a validation of models is essential. Additionally, there is only little support for writing OCL expressions for models. There are still tools that do not support OCL at all. For an adequate support each vendor has to implement the full functionality of OCL by itself, also those parts that manage the same functionality in every CASE tool, for instance the OCL parser. Due to these issues, our decision is a generic approach for resolving identical OCL functions, and adaptors that organize the transformations for different CASE tools.

Furthermore, there are many UML metamodel implementations – ArgoUML or Enterprise Architect, where the UML models defined by the developers are not validated against the well-formedness rules (WFR) (described with OCL) in the UML specification. An automatic validation of the OCL expressions in the UML specification before code generation allows a detection of errors in the early design phase and reduces problems for the further progress of projects. Therefore, it is of a huge value that UML tools support the definition and validation of OCL expressions.

An adaptable approach, called Adaptable OCL Engine (ADOCLE) for transforming OCL expressions could be reused in other environments or CASE tools (see Figure 2). Therefore the OCL expressions are applied to a selectable source metamodel, which is mapped to an arbitrary target metamodel. ADOCLE generates an equivalent expression of an OCL expression in the target metamodel depending on transformation patterns. The equivalent target metamodel expression is derived from the mapping between the source and target metamodel.

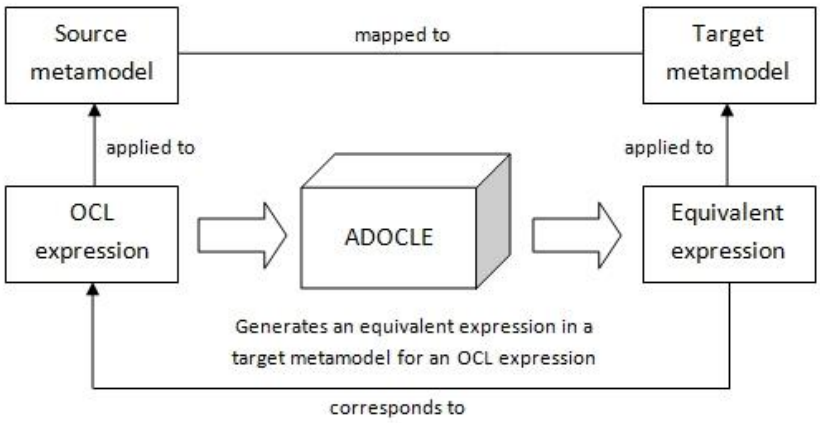

Figure 2 – Adaptable OCL Engine (ADOCLE)

[4] http://www.eclipse.org/

[5] http://www.eclipse.org/modeling/emf/

1.3 Result

The aim of this master's thesis is a prototypical implementation of ADOCLE (see Figure 3). We use the UML metamodel as source metamodel and the well-formedness OCL expressions in the UML specification. Enterprise Architect is chosen as a CASE tool, which uses a generic metamodel to provide the support of drawing models of different kinds. It is expressed as database schema and defines attributes, connectors, elements and operations of specific types depending on the supported kinds of models. Thus, the models are stored in the database. The target metamodel used for this master's thesis is the physical database schema of EA, that is described with Structured Query Language (SQL) and based on the SQL-92 standard [**7**]. The mapping between the UML metamodel and the EA database schema is based on patterns, which identifies the equivalent UML metamodel elements - the stored models in the database of the EA. The main task is to analyse the OCL expression and generate an equivalent SQL expression for the target metamodel – the database schema of EA. That means, deriving an SQL query semantically equivalent to the OCL expression to validate the model stored in the EA according to the UML metamodel.

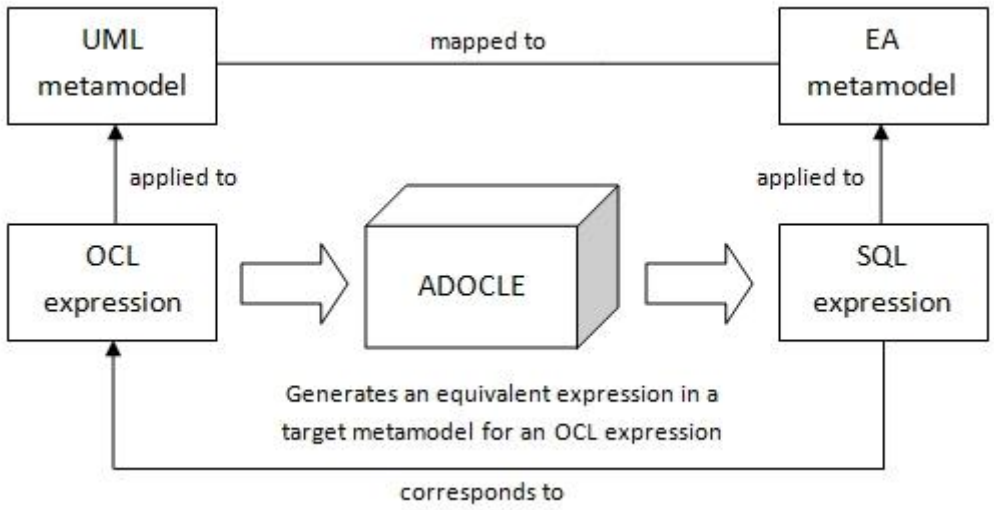

Figure 3 – Approach of ADOCLE prototype mapping between UML and EA

In this thesis, just a subset of the UML metamodel was selected. The chosen package is state machine, because this package contains a wide range of existing modelling paradigms and represents the complexity and power of the Unified Modelling Language. This also pertains to the OCL expressions in the UML Specification (see Chapter 4.2.3.5).

Enterprise Architect provides an API, which allows running programmatically through the model. This approach is manly used to provide validation rules within Enterprise Architect. It enables the definition of grained validation rules, which also considers tool specifics and additional information not contained in UML, but used in EA. However, writing such

rules requires the knowledge, how UML models are persisted in the EA database. Furthermore, using the EA API is not as fast as using SQL queries to access the database.

Therefore, a performance analysis has been conducted that compares the prototype of ADOCLE and the EA's approach for validating the model with the API. The evaluation shows the performance differences using SQL commands that are executed on the database – ADOCLE - and a specific realization of the OCL rules based on a source code driven solution – the EA API. A consistency analysis based on statistical hypothesis evaluate the

1.4 Methodical approach

The methodological approach for this master thesis consists of three parts:

1) **Tool and literature research:** One goal of the work is a prototypical implementation of ADOCLE, a research for CASE tools is done to define a general interface for mapping the metamodels of several CASE tools. Additionally an abstract approach for the metamodel mapping between the source and the target metamodel has been defined based on the metamodels of CASE tools.
2) **Implementation:** The prototype is implemented in a four layered architecture: OCL Metamodel Loader, OCL Parser, OCL Transformator and OCL Validator (see Chapter 4.2). The UML metamodel, the database schema of the Enterprise Architect and the mapping between is loaded to generate an equivalent expression in a target metamodel for an OCL expression resolved from an OCL parser. For the realization of the OCL2SQL transformation a literature research was done (see Chapter 3.3) to achieve a suitable solution. The OCL Validator is responsible for the automatic validation and lists model elements, which do not fulfil the OCL expressions.
3) **Evaluation:** As final step, the performance of ADOCLE prototype is evaluated and compared with a solution based on the EA API. The evaluation is a statistical hypothesis test that identifies false positives and true negatives for defined measurement parameters (see Chapter 5.2) to categorize failing executions for further implementation steps.

1.5 Structure of the work

This work consists of six further chapters.

Chapter 2 gives a general introduction in the Unified Modelling Language (UML) and Object Constraint Language (OCL). The definition, structure, main concepts and the area of application of the languages are discussed. Moreover, the advantages and disadvantages are compared and the usage is pictured by means of examples.

Chapter 3 focuses on the analogy of OCL and SQL. First, a short overview of the Structured Query Language (SQL) is given. After the contrast between OCL invariants and SQL queries is illustrated by means of the UML motivating example, the advantages and limitations are discussed. Furthermore, related approaches using mapping patterns are figured out in comparison to this work.

Chapter 4 explains a realization of the concept ADOCLE. First of all, an overview explains the module-based architecture and the communication between the components. Each module is depicted in detail to demonstrate the steps to generate an equivalent expression in a target metamodel for a given OCL expression. Diagrams picture the interaction between the components and examples show the intermediate results produced from the relevant module. The occurred problems and restrictions during the development process of the prototype are discussed in the corresponding subchapter.

In Chapter 5 we deal with the evaluation of the prototypical implementation of ADOCLE. A performance analysis of the executed OCL expression shows the time differences between the prototype and the solution using the EA-API. Finally, a statistical hypothesis test identifies false positives and true negatives for defined measurement parameters to categorize failing executions for further implementation steps.

Related work about other available published solutions as well as similar approaches are discussed in Chapter 6. The topics are divided in three categories: similar metamodel-based approaches for OCL compiler, the support of the most well-known OCL tools and applications and works concerned with verification and validation based on OCL.

The final Chapter 7 of this master's thesis gives a conclusion that summarizes the main points and gives an outlook on future work for ADOCLE.

2 Introduction of validating UML models using OCL expressions

This hapter deals with the standards: Unified Modelling Language and Object Constraint Language developed by the Object Management Group. We concentrate on aspects concerning the language definition and do not discuss building UML models of software systems. There are several books (see [**8**], [**9**]) that give an introduction to the UML techniques in more detail.

Chapter 2.1 explain the approach of UML, the definition and its notation. Furthermore, the abstract and the concrete syntax of the UML modelling concepts are commented. A motivation example illustrates the use of class diagrams and the application of OCL expressions. The example is frequently used throughout the remaining text for discussing various concepts and their formalization.

Chapter 2.2 gives an introduction of OCL. The structure and main concepts of OCL are figured out. Short examples illustrate the usage of UML models and OCL expressions, which shows the need for a more precise definition of UML models. Furthermore, the scope of applications of OCL and related languages is discussed.

2.1 Unified Modelling Language in a nutshell

At the beginning of the 1990, a number of modelling techniques for software systems came up. These object-oriented models are often defined by informal definitions expressed by proprietary notations. Software developers could exchange their ideas and decisions during the design phase of a software system using modelling techniques. But comparisons between informal languages are difficult due to different notation for the same concept. Sometimes a notation has different interpretations in different languages. UML originated from three leading object-oriented methods (Booch [**10**], Object Modelling Technique (OMT) [**11**] and Object-oriented software engineering (OOSE) [**12**]) and the incorporation of experience and best practices in software development. UML was a unification of these different approaches. Furthermore, it is based on a formal metamodel to define its structure and semantics.

The Object Management Group (OMG) defined UML as a graphical language for specifying, constructing, and documenting the artefacts of systems [**1**]. The Unified Modelling Language has been widely accepted as standard for modelling object-oriented software systems. It can be applied to a wide range of application domains and implementation platforms. The objective of UML is to provide users a common formal language to express their ideas, modelling techniques as well as best practises for every step of the development lifecycle of software

systems, from requirements analysis to maintenance. For instance, high level business problems may be expressed with UML as well as very low level real-time systems. The UML concepts are defined in a technology-independent manner within a MOF-based metamodel (see Chapter 2.1.1). Meta Object Facility (MOF) serves as meta-metamodel and defines the meta language. An explanation of the semantics of the UML concepts and a specification of human-readable notation for defining models corresponding to user's needs, can be found in the UML Specification [**1**], [**2**].

The current version of UML consists of four complementary *Requests For Proposals* (RFP):

1) **UML Infrastructure**
   The UML Infrastructure forms the base, which defines a reusable meta-language core, including UML, MOF and CWM. This core describes class-based language units, for instance, the concepts of classes, associations and properties encountered in most popular object-oriented programming languages. The architectural approach supports a fully interchange of models and contains a profile mechanism to adapt and customize modelling languages.

2) **UML Superstructure**
   The UML superstructure is based on the UML Infrastructure and can be seen as an actual definition of the UML that is well-known. It is structured modularly due to the wide range of application domains. UML user can select required parts and ignore other modelling concepts. For the ease of model interchange the UML concepts are partitioned into four horizontal levels, so called compliance levels. Each level adds new language units to extend the previous level:
   - Level 0 is specified in the UML Infrastructure and defines a class-based structure that provides a basis for interoperability between different modelling tools.
   - Level 1 is extended with language elements to support actions, activities, interactions, structures and use cases.
   - Level 2 adds units for deployment, profiles and state machine modelling.
   - Level 3 completes the UML with the concepts of information flows, templates and model packaging.

3) **UML Object Constraint Language (OCL)**
   OCL is a formal language developed by the Object Management Group (OMG) in parallel with the Unified Modelling Language (UML). It is often seen as a textual extension of UML, which specify conditions on UML models that must hold for the system or queries over objects defined in the model. Chapter 2.2 gives an introduction of OCL.

4) **UML Diagram Interchange**
   The UML Diagram Interchange is concerned with the exchange of UML diagram information. A specification based on XML Metadata Interchange (XMI) schema enables the interchange between UML tools.

2.1.1 Metamodel Layering

Like many natural languages (such as German or English), UML has its own grammar and their application rules. Whether it is a text- or model-based language, a precise definition of the language is necessary. Therefore, formal languages or metalanguages are used.
In computer science, formal languages have a long tradition. John Backus and Peter Naur invented a syntactic metalanguage for the definition of programming languages and for many other formal definitions. The syntactic metalanguage Extended Backus-Naur Form (EBNF) described in this standard [**13**] is based on Backus-Naur Form (BNF) and includes the most widely adopted notation for defining the syntax of a textual language by use of a number of rules.

However, model-driven engineering (MDE) aims to raise the level of abstraction in specification and increase automation in development. The idea promoted by MDE is to use models at different levels of abstraction for developing and interchange systems, thereby raising the level of abstraction. The abstract syntax of UML follows a metamodelling approach, which means that the same technique that is used for modelling application domains is used to define the UML itself. The abstract syntax of UML defines compact language units on a higher level, for instance, the concepts of classes, associations and properties for the reusability of these concepts in the different kinds of diagram models. The well-known techniques to models different perspectives of a system under development (see Chapter 2.1.2) in form of graphical notations represent the concrete syntax. In contrast to EBNF, the abstract and the concrete syntax are not part of the same metamodel layer, but both are defined using the terms of the same metamodel.

The OMG developed an approach for metamodel layering (depicted in Figure 4) to illustrate the different layers that always have to be taken in account when dealing with metamodelling. Each layer can be viewed independently of other layers. The concept of the four-layer hierarchy is based on following principles.

- The upper layer defines the base for the underlying layer.
- The underlying layer is an instance of the upper layer.

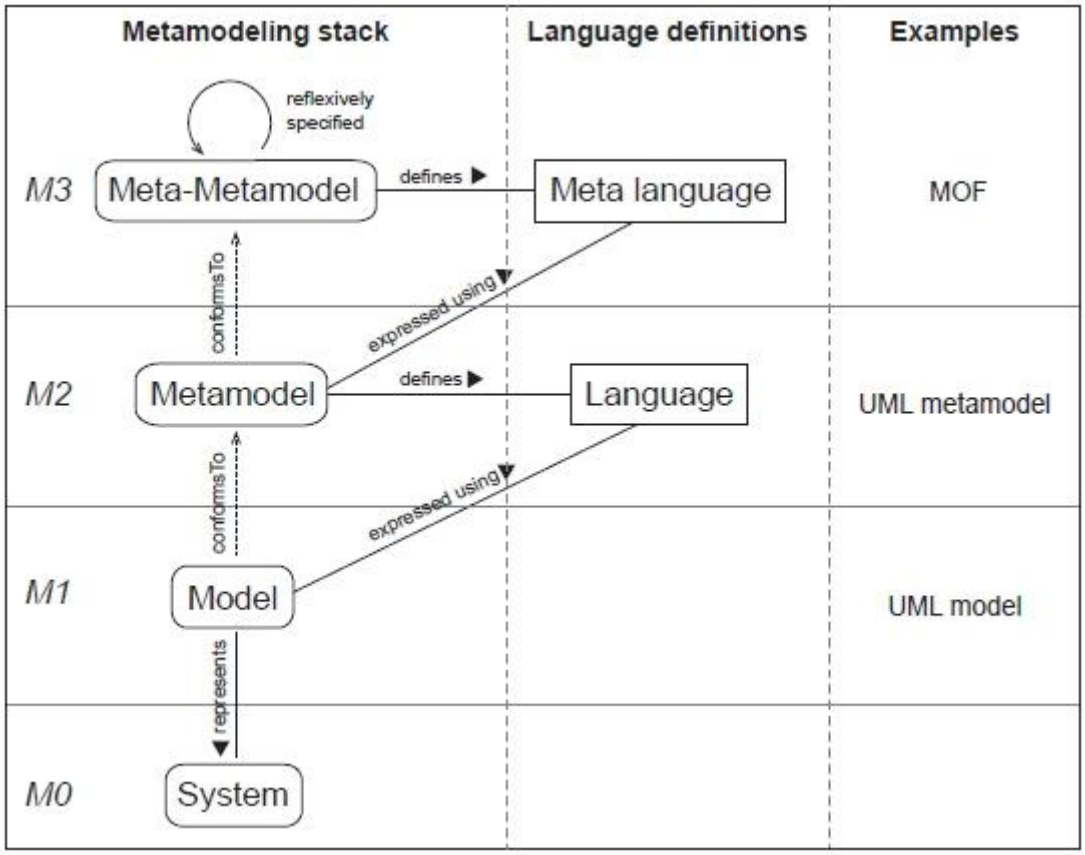

Figure 4 - Four-layer metamodel hierarchy

Meta-Metamodel

The meta-metamodelling layer, so called M3 forms the basis of the four-layer metamodel hierarchy. A meta-metamodel is a compact set of definitions that can be used to specify several metamodels. All model elements on the M2 layer are specified by means of meta-metamodel elements. In other words, every metamodel conforms to some meta-metamodel. In the case of the UML metamodel the Meta Object Facility (MOF) serves as meta-metamodel and defines the meta language. The four-layer hierarchy ends with the layer M3, because the MOF is reflective. That means that this language can be defined by itself and needs no higher layer.

Metamodel

The metamodel layer, often referred as M2 layer is responsible for the definition of a language for specifying models. The concepts and constructs of a metamodel conform to the meta-metamodel. In the case of the UML metamodel, it is an instance of MOF, the concepts are specified by class diagrams that define properties and relationships between model elements. In earlier version of UML conditions of the model elements were often defined in natural language. Additional constraints are required to restrict the set of legal UML models. Gradually, the UML Specification is extended by well-formedness rules (WFR) expressed as OCL invariants on model and metamodel elements.

Model

The term model is very widespread and has different meanings depending on the application field. In general, a model captures a view of a system. It is an abstraction of the reality, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the system that are relevant to the purpose of the model, at the appropriate level of detail. In the paper [**14**] a discussion about the definition, the meaning and the interpretation of the term model is found. The author defined the term model as “a set of statements about some system under study.” However, there are a large number of definitions that have in common, that a model, in context of software development, is used to represent a software system. In the four-layer metamodel hierarchy a model is situated on the layer M1, as an instance of the metamodel. UML models are used to describe structural or behavioural aspects of software systems of a wide range of application domains, such as requirements, the architecture or the user interface behaviour of software systems.

Runtime instances or snapshots

At the bottom of the four-layer hierarchy, the runtime instances or snapshots of the system can be found.

2.1.2 Notation

As mentioned above, UML provides the ability to model different perspectives of a system. Therefore thirteen different diagram types can be chosen to illustrate the complexity of the system. The diagram types can be categorized into diagrams focusing on structural or behavioural aspects of a software system. The following sections give a brief summary of the most important diagram types of UML.

2.1.2.1 Structure diagrams

Structure diagrams allow capturing the structure starting from a single class up to a specification of a complete architecture of systems. Each diagram type focuses on another perspective of the structure of a system.

Class diagram

The class diagram is used to depict the structure of data and its behaviour in an object-oriented manner. It describes the essential parts like classes, attributes and relations to each other. The class diagram is also used to define the fundamental modelling constructs of UML that represent the core of the modelling language.

Package diagram

The package diagram enables grouping the structure of arbitrary systems and describing the dependencies between them.

Object diagram

The object diagram describes a snapshot of the defined system during the runtime, represented by instances of classes, components, associations and attributes.

Component diagram

The component diagram allows modelling the components involved in a system and their dependencies. Components are modular parts of a system that provide access through clearly defined interfaces to the behaviour of the component.

Composite structure diagram

A composite structure diagram shows the internal structure and interactions of a classifier. A classifier is an abstract element of the UML metamodel. Thus, for example, the model elements class, interface, component, behaviour, activity, interaction or state machine are specializations of the classifier.

2.1.2.2 Behaviour diagrams

Apart from modelling the given structure of a system, it is also essential to capture its behaviour. Behaviour diagrams represent behavioural specifications from different perspectives emphasizing or ignoring certain aspects and complement each other to the sum up of a more or less complete description of the overall behaviour.

Use Case diagram

A use case diagram shows the system from the perspective of the end-user. The functionality of a system is described in terms of use cases. Interactions of the users, also called actors, and the relationships between actors and use cases are captured in a use case diagram.

Activity diagram

An activity diagrams is a notation instrument for modelling any kind of process. Using this type of diagram, a complex progress, including any concurrencies, alternative decisions or similar behaviour can be modelled and reconstructed concretely. It describes the control as well as the data flow between the actions of a system.

State machine

A state machine is used to model the behaviour of a classifier of any kind. The internal behaviour of the classifier, for instance a class, is described in terms of states that the classifier can assume, the transitions between the states and internal or external events.

Sequence diagram

Sequence diagrams illustrate interactions within structural elements. An interaction is an exchange of information between two or more communication partners. The main focus of a sequence diagram is to specify a chronological order of the interactions.

Communication diagram

The communication diagram is very similar to the Sequence diagram. The communication between objects in terms of interactions is modelled, but this diagram type illustrates the interaction of the communication partner on a leverage level of abstraction.

Timing diagram

Timing diagrams are an additional tool for describing interactions between objects. It is originated by the electrical engineering community and looks like a graph of an oscilloscope. It enables more precise temporal specifications than the other UML diagrams, and is therefore most appropriate for the design of real-time systems.

Interaction overview diagram

This kind of diagram visualizes the order of interactions and the conditions that has to be solved to start an interaction. It is a hybrid of activity diagram and interaction diagrams.

2.1.3 Advantages and disadvantages

The advantages of a graphical modelling notation are obvious. After a short explanation about the notation diagrams are easy to understand, even by non-experts. The visual illustration of the structural as well as behavioural aspects of a system highlights the relations between the objects in an intuitive manner that is not available in purely textual notations. Nevertheless, there are several problems related to graphical languages. One of the major problems is the wide interpretation room of the UML semantics. The semantics of UML which describes the meaning of the language elements is only informally defined in natural languages. An imprecise specification can lead to several problems [**15**]. Due to the wide interpretation room of the UML semantics, inconsistencies and ambiguities are the consequences. Misinterpretation of UML models in software development teams can lead to inconsistencies of components and communication problems and pose an additional burden for the team. In such cases the modelling phase of the development process needs more time and effort to communicate the interpretations.

The imprecise specification of the semantics of UML also plays a significant role for the implementation of tools that supports the techniques of UML. The different tool vendors develop the syntax and the semantics of the UML in their own manner, which results in interoperability problems. So called semantic variation points provide additional inconsistencies for the interoperability of models between different vendor solutions. Semantic variation points are those parts of the UML Specification that are in aware not completely refined to allow the domain-specific extensions more flexibility to define additional components.

Without precise semantics definition it is also hard to validate or verify UML models formally. UML is specified by means of the UML class diagram notation. Thus, UML builds a formal language that can be validated. But a UML class diagram has its limitations and is not expressive enough to provide all relevant aspects that can be formulated in natural language. The following example may illustrate the expressive limits and why the current version of UML is extend by the Object Constraint Language (OCL). Therefore, a class diagram in Figure 5 shows the static aspects of a simple system of vehicles and their components on layer M1. A vehicle has an engine and wheels. An engine also contains wheels. Due to the simplicity of the model, it is possible that an instance of a wheel is part of a vehicle as well as of an engine. Of course, the model can be extended with specific types of wheels (gear wheel and driving wheel) and set the relation in a correct manner. When we consider the complexity of a car, the model will grow enormous according to the high number of specific types. A constraint that does not allow the mentioned behaviour could solve the problem without any extensions and the model could be kept simple.

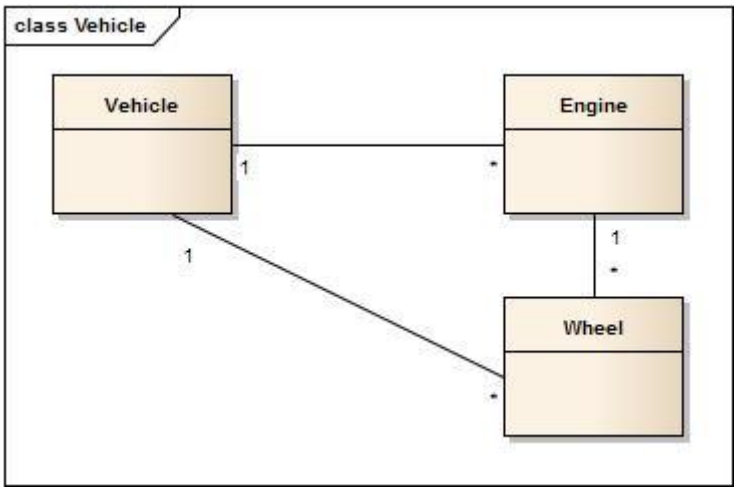

Figure 5 - Static aspects of UML

UML diagrams provide the ability to attach constraints as notes to any graphical UML element. In particular, UML class diagrams use such constraints to annotate pre and post conditions as well as class invariants. Therefore, OCL can be a solution. OCL expression can specify conditions on UML models that must hold for the system. The example describes a possible solution using OCL expressions.

```
-- A wheel of an engine cannot be a wheel of a vehicle
context Vehicle inv :
self.engine.wheel->select( w : Wheel | self.wheel != w)
```

In the case of UML metamodel defined by UML class notations, OCL expressions improved earlier versions of UML by reformulating the well-formedness rules in the UML Specification and add a higher level of preciseness. In Chapter 2.2 we discuss the usage of OCL in detail.

2.1.4 Motivation example

For a better understanding, we use the following motivation example throughout this master thesis to demonstrate the concepts of UML and the need of OCL. Examples for OCL expressions are described in the following Chapter 2.2 regarding to the class diagram in Figure 6. In the rest of the thesis OCL is used on the metamodel layer (M2) However, to get an idea how OCL can be applied, a simple UML class diagram model on layer M1 is used.

Figure 6 depicts a similar example like within the OCL specification [**3**], to describe the main constructs of OCL. Additional parts are used for the demonstration of not involved UML concepts like the generalization and for easier explanations of the concepts and constructs. The class diagram uses the following UML concepts: Classes (Person, Company, Account, Vehicle, etc.) with attributes and operations, associations, generalization (Vehicle is a generalization of Car and Motorcycle), two reflexive associations (Person), and multiple as-

sociations between classes (Person and Company). Association ends are adorned with role names and multiplicity specifications.

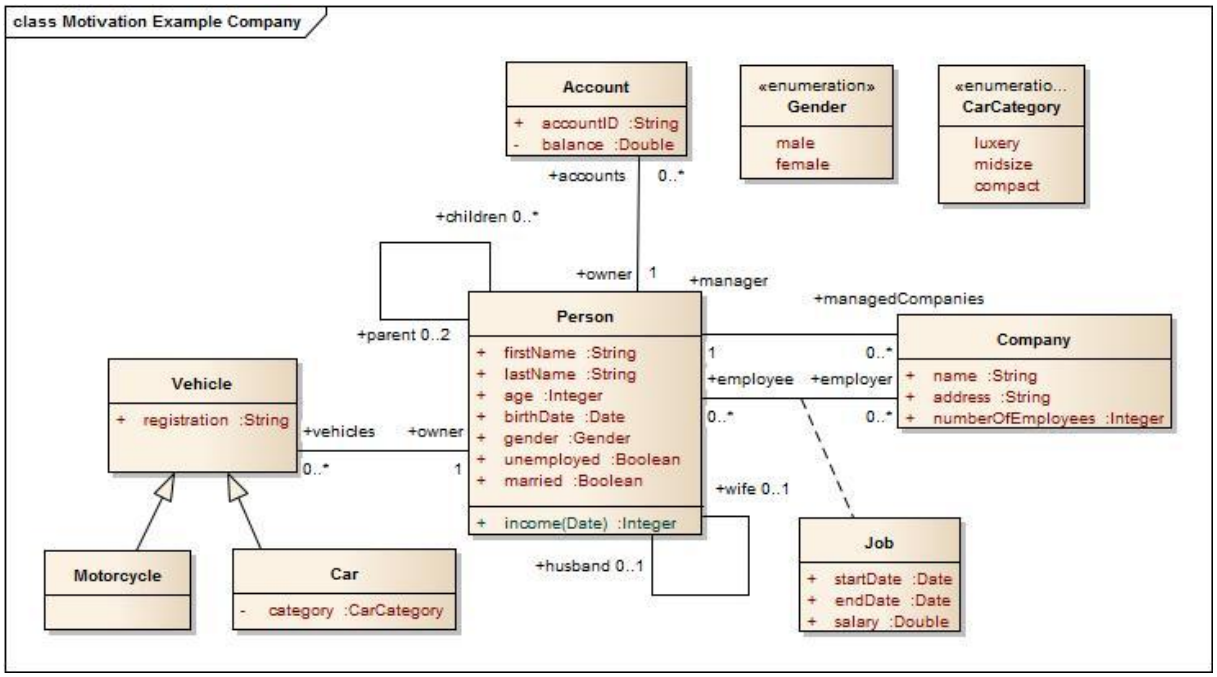

Figure 6 - Motivation example - UML class diagram

A simple example for the need of OCL expressions is the attribute age in class Person. Without further constraints, a person may have a negative value for its age. The following expression shows a condition for persons that only positive values for the attribute age are allowed.

```
-- The age attribute of persons has to be greater than zero.
context Person inv expressionForAttributeAge:
self.age > 0
```

As mentioned above, the expressiveness of UML class diagrams allows restricting possible combinations of instances, but lack of the possibilities to define more complex constraints. For instance, a class Person may have two reflexive relations to itself (biological child). On one end of the association the father is called, else on the other the mother is determined. Both have the multiplicity of one. The other direction of both associations has the role name children with the multiplicity of *. The class Person has the attribute gender. With this class diagram different family trees may be expressed, but it is impossible that one person has a father and a mother with the same gender. Biological such constructs are impossible. Such instantiations has to be restricted. Further rules have to be defined on the model. OCL expressions can be used to specify such conditions.

These are just two examples that demonstrate the benefits of using OCL. More complex OCL expressions are discussed in the next chapter.

2.2 Object Constraint Language in a nutshell

The Object Constraint Language (OCL) is a formal language developed by the Object Management Group (OMG) in parallel with the Unified Modelling Language (UML). It is often seen as a textual extension of UML, which facilitates to specify conditions on UML models that must hold for the system or queries over objects defined in the model.

A UML model has its limitations and is in general not expressive enough to provide all relevant aspects of a specification of a software system. One simple approach is to provide the restrictions in natural language. However, this may lead to ambiguities, no automatic validation and no automatic code generation is possible. Typically OCL expressions are used to specify constraints on models that cannot be expressed, or are very difficult to express, with the UML notation. Queries defined by OCL expression are completely programming language independent and can be integrated in the code generation process or may be used in model transformations. There are no side-effects when the objects are evaluated.

OCL is based on a first-order predicate logic. In contrast to other formal specification languages like Vienna Development Method (VDM) [**16**], Z-Notation [**17**] or B-Method [**18**] (see Chapter 2.2.6) the syntax of OCL resembles a programming language than a first-order predicate logic. But it is not a programming language, just a **pure specification language**. Previous designed formal specification languages are very hard to learn and could not prevail in the industry. So they are mostly used in academic world. The intuitive syntax of OCL is closely related to the syntax of UML that is widely accepted as standard for modelling object-oriented systems and provides a more adequate every-day modelling than a pure first-order predicate logic for developers. The language provides variables and operations which can be combined in various ways to build expressions. Frequently used language features are accessing object's attributes, navigations to objects that are connected via association links, and operation calls. OCL is an **expression language** that guarantees no side effects; it cannot change anything in the model. Whenever an expression is evaluated, it simply returns a value of type that conforms to the OCL type hierarchy. OCL defines a number of data types including basic type such as Boolean and Integer, as well as types for dealing with collections. OCL is also a **typed language**. That means each object, attribute, result of an operation or navigation has a specific type. An OCL expression must conform to the conformance rules of OCL to be well formed, independent if it is a basic type, OCL-specific type or user-specific type.

2.2.1 Scope of application of OCL

OCL can be used for different purposes: Whether as a constraint language for the specification of model definitions or as a query language for models. Aspects that cannot be expressed or are very hard to define by means of UML notations use constraints, to specify the

requirements of object-oriented systems. These kinds are distinguished in 6 categories of constraints:

1) Invariants
2) Pre and post conditions
3) Initial and derived values
4) Guards
5) Definition of attributes
6) Body definition of operations

The primary task of OCL is the formulation of invariants in terms of restrictions on the validity of object models and pre and post conditions of operations. An invariant is a constraint that should be true for an object during its entire lifetime. Additionally, pre and post conditions enable behavioural specifications of operations in terms of conditions on a previous state and a post-state after executing the operation. The other categories extend the OCL with comfortable auxiliary tools and are explained in the OCL specification [**3**].
OCL is a language on its own and can be used for different models on different layers (M1, M2, and M3) in the metamodel hierarchy. The previous example in Figure 6 illustrated the use of OCL for models on layer M1. But, OCL enables also the specification of well-formedness rules (WFR) for UML models on the metamodel level (M2). For example, the UML metamodel contains OCL expressions that explain the complex behaviour between subcomponents within or across the packages of the UML metamodel. This copes with the limitation of the UML class diagrams and helps to define more restrictions how models on the layer M1 can be created and validated.

Because of the descriptive nature of OCL, expressions can also be used for specifying queries. Queries allow the developers to navigate and inspect objects and data interactively. Especially, in the case of database applications queries are a useful feature when dealing with large result sets of objects. Considering OCL as query language, the scope of application concentrates on model or metamodel transformations and code generation. The OCL expression helps to specify the transformation patterns. The result sets of the source and the target model or metamodel elements that should be transformed are defined with OCL queries.

As briefly mentioned, there exists an analogy to other query languages, for example, SQL for relational database management systems or XPath/XQueries for XML documents that we explain in Chapter 3 in detail.

2.2.2 Definition of constraint

Before we can go further with the core concepts of OCL, the term constraint has to be defined. According to Warmer and Kleppe [**19**], “A constraint is a restriction on one or more values of (part of) an object-oriented model or system“. Constraints may be denoted within the graphical illustration or in a separate document.

2.2.3 OCL core concepts

This chapter gives a briefly summary of the concepts of OCL.

Context

The context could be interpreted as entry point to a model element where an expression is attached. The keyword *context*, followed by a name of a model element, typically a name of a class or interface, represents this entry point. Keywords *inv*, *pre* or *post* denotes the stereotype of the OCL expression: invariants, pre or post conditions, followed by an optional name for the OCL expression. In the case of invariants, a classifier is associated and the result type is a Boolean. For pre or post conditions additionally an operation signature is given and variables may be defined (see examples below). The actual expression comes after the colon. Therefore the reserved word *self* can be used to refer to the contextual classifier in an OCL expression.

Objects and properties

As illustrated in the motivation example in Figure 6 the class diagram contains other classifier than classes or interfaces, e.g. attributes, operations and associations. An OCL expression can refer to all these classifiers without side effects. The OCL specification [**3**] speaks from accessing different kind of properties. A property is one of: an attribute, an association end or an operation (method) which may be referred using a dot notation followed by the property name.

Properties: Attribute

In the following example, the keyword *self* refers to an object of class *Person*. An attribute of an object, such as the *unemployed* attribute may look like as follows.

```
-- The access to the attribute unemployed of the class person
context Person inv:
self.unemployed
```

The OCL expression describes, if a person is unemployed or not. Therefore the attribute unemployed is defined with a basic type *Boolean* for specifying an either-or-decision. Attributes on basic value types (see Chapter 2.2.4) can express calculation over the class model. An instance is shown in the section invariants. Attributes may have other types than basic types which are explained in Chapter 2.2.4.

Properties: Operations

Operations can be accessed as attributes. In contrast to attributes, operations may have parameters and a return type. The result of an operation is a value of the return type, if a return type is set. Otherwise the return type has to be defined with *void*. In Figure 6 the class *Person* has an *income* expressed as a function of the date. In the example, the *income* of each instance of *Person* on the key date: 12-12-2012 is evaluated. The definition of the operation income is shown in the section pre and post conditions below.

```
-- The access to the operation income at the key date: 12-12-2012
context Person
self.income(12-12-2012)
```

Properties: Association-ends and navigation

A navigation is a reference from one object to another object (or to itself) using the name of the opposite association-end (role name of an association). If the role name is missing, the name of the type of the opposite association-end – direct reference to the other object - can be used instead. The value of the expression is dependent on the multiplicity of the association end. When the maximum of the association-end is one, the value of this expression is an instance of an object. Otherwise, a collection of the type of the object of the referred association-end is returned. The example in Figure 7 illustrates some kind of navigations between the classes *Person* and *Company*.

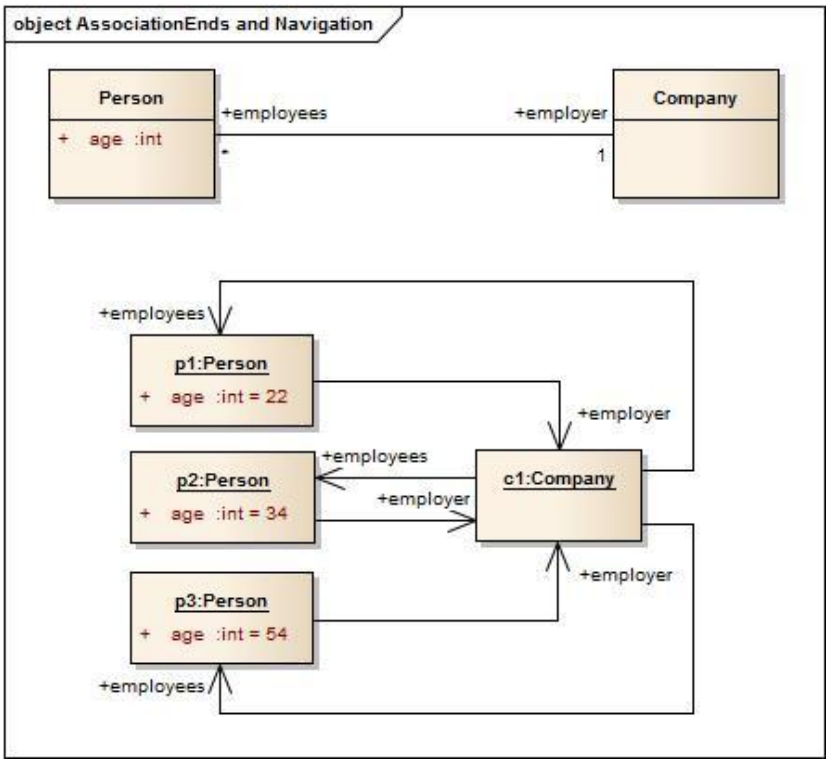

Figure 7 - Association-end and Navigation

```
-- Navigation from Person to Company
context Person inv: oneEmployer
self.employer

-- Navigation from Company to Person
context Company inv: listOfEmployees
self.employees
```

The first OCL expression shows the navigation from the object *Person* to the object *Company* using the role name *employer*. In the example, the result is the object *c1* with the type *Company*, as pictured in Figure 7. The second OCL expression illustrate the navigation in the other direction, starting from the object *Company*, navigating to the object *Person*. Due to the multiplicity at the association-end named *employees,* a collection of objects of Persons is returned. In the example, the result is the Set {p1, p2, p3} with the type Set {Person}. By default, navigations will return a Set.

Other predefined collection types in OCL like Bags, OrderedSet or Sequences are described in Chapter 2.2.4. Each collection type provides a number of OCL operations, as well as every type in the type hierarchy. The OCL operations are not comparable with the operations described above. The operations within the example class diagram are part of the problem specification and represents methods for managing them. The OCL operations are functions to enable a flexible and powerful way of projecting modified collections from existing ones. Generally, within OCL expressions, properties and operations are separated by a dot character depending on the type hierarchy. In the case of collection type an arrow '->' followed by the name of the operation is used.

```
-- number of employees
context Company inv: listOfEmployees
self.employees->size()
```

The OCL expression calculates the number of employees. For the example, pictured in Figure 7, the result is 3. The type of the operation *size* is the basic type *Integer*.

Invariants

An invariant is an OCL expression that express rules with the type Boolean applied to a specified classifier. That means that an invariant must be true for all instances of a specific type (class, association class or interface) in the model, which is the context of that constraint.

```
-- The age attribute of persons has to be greater than zero.
context Person inv expressionForAttributeAge:
self.age > 0
```

The given example expresses that only positive values for the attribute age are allowed. As a consequence only persons with an age greater than zero are valid. The invariant is true, if every binding of self to an object of a class *Person* and its attribute *age* are greater than zero. In most cases, the keyword *self* is enough to refer to a classifier. As alternative, a different name can be used to play the role of self.

```
-- The age attribute of persons has to be greater than zero.
context p:Person inv alternativeExpressionForAttributeAge:
p.age > 0
```

Pre and post conditions

OCL also supports the definition of pre and post conditions that form contracts for operations. Pre and post conditions are assertions that must be true either before or after the body of the context operation executes. The following example illustrates both kinds of conditions on an operation called *income*.

```
-- The pre and post condition for the operation income.
context Person::income(d : Date) : Real
pre: self.job.startDate < d and self.job.endDate->isEmpty()
post: result = self.job.salary->sum()
```

The operation calculates the current income of a person. The pre condition checks, if the given date is after the startDate and the endDate of the job is not already set. The post condition totalizes the salary of each job where the pre condition is true. Other categories like the initial or derived values work in a similar way. Detailed explanations can be found in the standard of OCL [**3**].

2.2.4 Types

OCL is a strongly typed language that means to each object, attribute, result of an operation of navigation a specific type is assigned. There are predefined value types that are part of the definition of OCL and each of the value types supports a set of applicable operations.

Basic value types

The basic value types of OCL are *Boolean*, *Integer*, *Real* and *String*. Examples for values and operations are listed in the Table 1.

Type	Values	Operations
Boolean	true, false	and, or, xor, not, implies
Integer	1, -5, 2, 34, 26524, ...	*, +, -, /
Real	1.5, 3.14, ...	*, +, -, /
String	'To be or not to be...'	concat(), size(), substring()

Table 1 - OCL basic types

Collection types

OCL provides a small set of collection types distinguished by whether or not the elements are ordered and whether or not they allow duplicates. The collection types listed in Table 2 have a common super-type Collection(T) – instance of CollectionType - and conforms to it. Collections are parameterized with a type parameter T that denotes the type of the elements within the collection. The type represented by the parameter T must hold the conformance rules based on a type hierarchy defined in OCL.

Collection Type	Description	Values	Type
Set	Mathematical set that does not contain duplicates and have no defined order.	Set{ Set{1}, Set{3,2} }	Set(Set(Integer))
Bag	Multi set that allow duplicates and have no defined order.	Bag{1, 2.0, 2, 3.0, 3.0, 3}	Bag(Real)
Sequence	The sequence is a bag with ordered elements.	Sequence{ 1, 2, 2, 2, 45, 60, 81}	Sequence(Integer)
OrderedSet	This collection type represents a set with order elements.	OrderedSet{'a', 'b', 'c', 'x', 'y', 'z'}	OrderedSet(String)

Table 2 – OCL Collection types

Tuple types

It is also possible to define tuples that compose several values. Tuples are a fundamental concept in most object-oriented data models (e.g., extended Entity-Relationship (EER) [**20**]) models) and logical data models (e.g., the relational data models). A tuple is an ordered list of elements, where each part of the list has its own type. It is required for expressing structured and complex queries. The following example shows a simple constellation of tuple presenting a pair of *Integer* and *Boolean* values. Complex examples can be found in the OCL Specification [**3**].

```
Tuple{ x = 5, y = false } : Tuple{ x : Integer, y : Boolean}
```

User-specific types

In addition to predefined types, user-specific types can be integrated in the OCL expressions through a model. Each instance of a class of MOF is automatically an allowed type. As mentioned above, an OCL expression is written in a context of a model. In the case of UML the context represents a classifier of the UML metamodel, which is an indirect instance of class of MOF. So each model element in an UML model is a user-specific type that can be used in OCL expressions. Also defined enumeration types are allowed. In the motivation example shown in Figure 6, an enumeration is specified in UML notation to define the gender of a person. The literals 'male' and 'female' could be integrated in OCL as follows:

```
context Person inv: gender = Gender::male
```

Type Hierarchy

Finally, OCL provides conventional oriented types or special types for organizing a type hierarchy (see Figure 8) and applicable operations on the predefined types. Object-oriented concepts such as inheritance, polymorphism, and strong-typing in object-oriented languages form the fundament of the type hierarchy. A subtype relationship induces a partial order on OCL types, while ad-hoc polymorphism guarantees an overloading of operations and parameterizing the element type of collection types. The work in [**21**] gives an easy introduction for more details in the concepts of object-oriented languages.

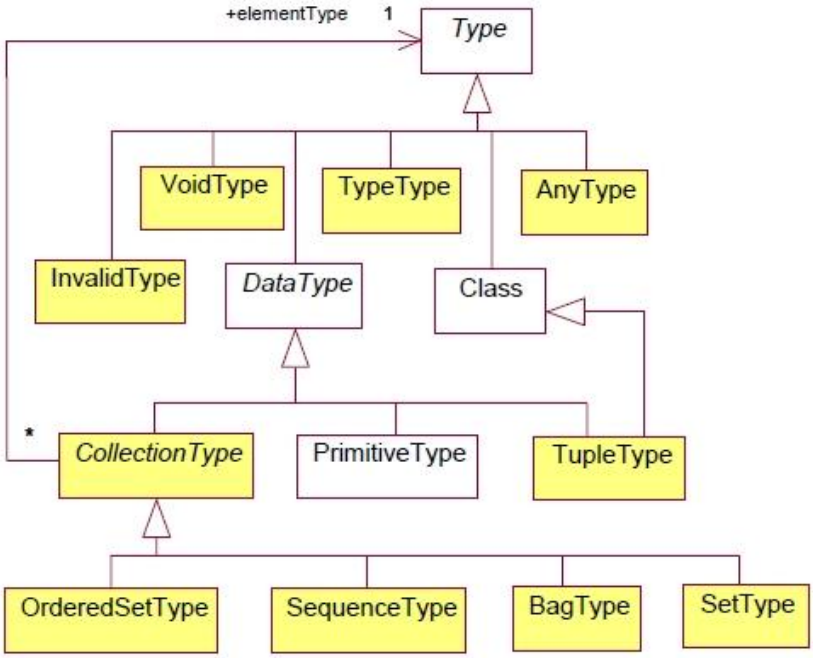

Figure 8 - Type Hierarchy for OCL

Special types

The type OCLAny is an instance of AnyType and represents the super-type of any type in OCL except for the collection and tuple types. In other words all OCL types conform to the type OCLAny. Each type within the type hierarchy supports a set of applicable operations. Due to the type hierarchy a subtype inherit the applicable operations of the super-type and their behaviour.

We briefly explain the type hierarchy concept of OCL by means of the CollectionType. The CollectionType is an abstract super-type of collection types in the OCL and defines the properties and operations on collections that have identical semantics for all collection subtypes, for instance, the operations *sum* and *size*. Some operations may be specialized in the subtype, for example the operation *count*. The parameterization of collections with a type parameter T is defined with an association to the interface *Type*. Note that there is no restriction for the elements of the parameter T. This means that a collection type may be parameterized with tuple types or other collection types.

The type OclType is a singleton instance of TypeType and provides access to the meta-level of a model. OclUndefined represents the type of a model element whose value has not been initialized. This type is often needed in model transformation where it cannot be assumed that all model elements have been initialized. For the identification of operations that does not have a return type, OclVoid can be used.

Conformance Rules

The type hierarchy determines conformance of the different types and specifies if the OCL expression is valid or not. For example, a comparison between a *Boolean* and a *String* is invalid. The type conformance is defined by the following conformance rules [**3**]:

- *Type1* conforms to *Type2* when they are identical.
- *Type1* conforms to *Type2* when it is a subtype of *Type2*. In other words, *Type1* conforms to *Type2* if an instance of *Type1* can be substituted at each place where an instance of *Type2* is expected.
- Type conformance is transitive. The type conformance is a relation that mirrors the subtype relation introduced by the type hierarchy. As a consequence, each type conforms to each of its supertypes.
- A parameterized type T(X) conforms to T(Y) if X conforms to Y. For example, Collection(Integer) conforms to Collection(Real), because *Integer* is a subtype of *Real*.

The generalization hierarchy of UML models are incorporated in the type hierarchy of OCL and follows the above conformance rules.

2.2.5 Expression

A typical OCL expression may look like the following:

```
context Person inv: self.married and self.unemployed
```

This expression defines that the *and* statement is true if both sub expressions *self.married* and *self.unemployed* are true. This example illustrates two fundamental characteristic of expressions:

1) Expressions can contain expressions as sub elements. The *and* expression is a non-terminal expression that contains a left operand (*self.married*) and a right operand (*self.unemployed*). Each operand of the *and* expression in the example represents an expression that contains two operands separated by the dot. The example shows that expressions must be generalized from some common abstract expressions to support polymorphism.
2) As mentioned in Chapter 2.2.4, every construct in OCL has a type, also expression own one. The left and the right operands are of the standard OCL type Boolean. As a consequence, the *and* expression returns true or false.

Figure 9 shows the basic structure of the abstract syntax of the kernel for expressions which is just an excerpt of the OCL expression package. The complete specification can be found in Chapter 8.3 in [**3**]. The abstract syntax is responsible for the inheritance relationships and the relations between the components.

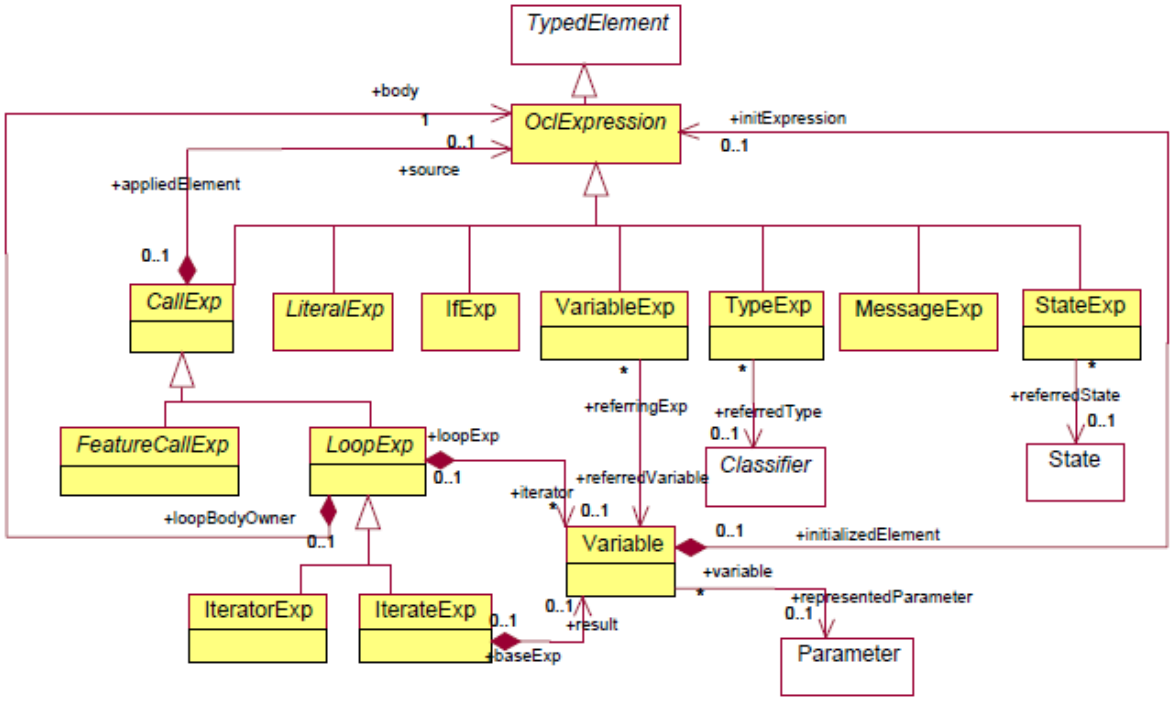

Figure 9 - Excerpt of the OCL expression package

The concrete syntax realizes the abstract approach in the form of a full attribute grammar. Every production rule is denoted using the EBNF formalism and annotated with synthesized and inherited attributes. The result of synthesized attributes, representing the left hand side of the production rules is derived from the attributes of the right part of the production rule. In addition, each production may have inherited attributes attached to it. Inherited attributes describe the environment of the production rule. The mapping between the abstract and concrete syntax is also part of the grammar. Therefore a synthesized attribute called *ast* is added to each production which has the corresponding metaclass from the abstract syntax as its type. The following production rule *OclExpressionCS* illustrates an example of the attributes. It defines the mapping to the abstract syntax component OclExpression and specifies the inheritance relationships shown in Figure 9.

Abstract syntax mapping
OclExpressionCS.ast : OclExpression

Synthesized attributes
[A] OclExpressionCS.ast = PropertyCallExpCS.ast
[B] OclExpressionCS.ast = VariableExpCS.ast
[C] OclExpressionCS.ast = LiteralExpCS.ast
[D] OclExpressionCS.ast = LetExpCS.ast
[E] OclExpressionCS.ast = OclMessageExpCS.ast
[F] OclExpressionCS.ast = IfExpCS.ast

Inherited attributes
[A] PropertyCallExpCS.env= OclExpressionCS.env
[B] VariableExpCS.env= OclExpressionCS.env
[C] LiteralExpCS.env= OclExpressionCS.env
[D] LetExpCS.env= OclExpressionCS.env
[E] OclMessageExpCS.env= OclExpressionCS.env
[F] IfExpCS.env= OclExpressionCS.env

Such production rules are the initial point for the OCL Parser described in Chapter 4.2.2. We used the EBNF of OCL to generate a parser generator for interpreting given OCL expressions. For the sake of completeness, an example (see Figure 10) pictures some kind of expressions to introduce the usage of expressions.

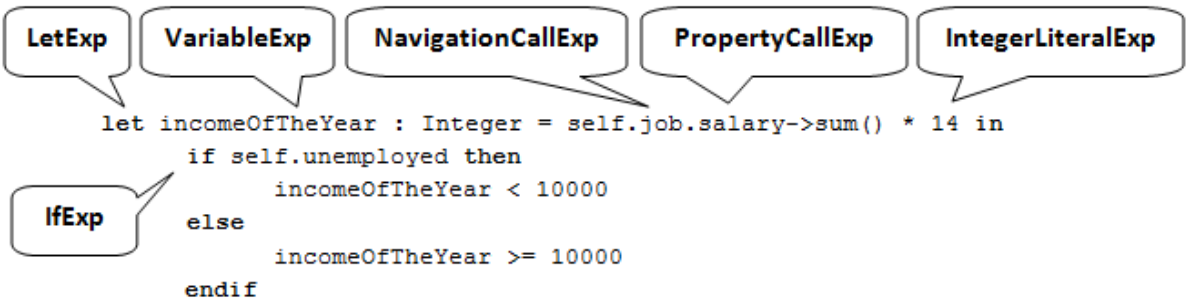

Figure 10 - Example of expression types

2.2.6 History and related languages

Originally, the Object Constraint Language (OCL) was developed by IBM[6] in 1995 with the purpose to define business models. During 1997 OCL was adopted as a formal specification language within UML 1.1. It is used to help formalize the semantics of the language itself and to provide a facility to precise models using constraints and well-formendness rules.

Jos Warmer, the developer of the originally language was inspired by the Syntropy method of Steve Cook and John Daniels. Syntropy is an object-oriented analysis and design method with the goal to provide modelling techniques that allow precise specification and separation of different areas of concern. It is based on at this time favoured graphical notations of OMT, combined with additional formal specification elements derived from the Z notation. The Syntropy, as described in the book [**22**], is not a complete method; the development on the approach stopped, maybe due to the complexity of Z notation. But many of the ideas are incorporated in the UML specification and other development processes. So Syntropy can be seen as a direct ancestor of OCL.

The formal specification language Z is grounded in mathematics – set theory and first-order predicate logic [**17**]. The set theory contains concepts for standard set operators, set comprehensions, Cartesian products, power sets and many more. Predicate logic constitutes a family of logical systems, making it possible to formalize arguments and to check their validity. The combination of the concepts forms the mathematical language of Z. It can be used to extend the previous object-oriented modelling concepts by a clear declaration of objects, values and types. Concepts like navigation expressions, various kinds of constraints pre and post conditions are provided to more precise semantic underpinning.

Fundamental parts of many formal methods were found in the late 60's at the IBM laboratory developed in Vienna. The result is known as the VDM (Vienna Development Method) [**16**]. The methodological approach has found a wide spread, especially the University of Manchester and the Technical University of Denmark continue the development. Already, VDM uses types and collections as well as classes and inheritance to specify refinements on models. It is still widely used in industry.

As alternative, Alloy can be chosen. It is a specification heavily influenced by Z. Alloy is first-order based which makes it automatically analyzable. The improvement in Z is rather more limited. In [**23**], the authors compare Alloy with OCL and show translations for a subset of the UML metamodel with well-formedness rules into Alloy. Alloy is not used as much in industry as VDM.

A big disadvantage of formal specification languages is that the mathematical description of constraints is difficult to learn and could not prevail in the industry. It's more of a language for mathematicians.

[6] http://www.ibm.com

Another similar language is the Extended Entity-Relationship (EER) modelling language. It is based on the Entity-Relationship (ER) model which is an abstract way of describing database designs. The EER model is an extension of the ER model including concepts of inheritance and polymorphism. Furthermore, it introduces the concept of a union type. The intention of EER was to determine more precisely properties and constraints for database concepts. The semantics and the fundamental paradigms of Extended Entity-Relationship model are presented in [**20**], [**24**]. In contrast to OCL, both languages provide a specification of declarative constraints and allow the definition of queries. The EER calculus is based on the set theory which guarantees a complete formal semantic. It is proved to be safe in the sense that all expressions yield a finite result. A proof of the first versions of OCL would lead to invalid states. For example, the OCL expression *Integer.allInstances* does not return a finite set of all Integer values. On the other hand OCL allows expressing navigations through class models using the association-end names. The readability of the expressions is enhances in most cases in comparison to the EER calculus that use SQL statements. A complete comparison of OCL and the EER calculus are illustrated by example in [**25**]. The relation of OCL and SQL for query expression is also included in this work. A more detail demonstration is shown in Chapter 3.

2.3 Summary

This chapter introduced in the UML definition, its purpose and the concepts are shown briefly. The formal metamodelling approach of UML provides a more precise definition than previous modelling languages and easy understandable graphical notations, also for non-experts. Nevertheless, it is a compromise between formality and informality. The missing formal specification for the UML semantics plays a significant role for the implementation and lets a wide interpretation room for the vendors. OCL can be a solution. OCL expressions improved the earlier versions of UML by reformulating the well-formedness rules in the UML specification and add a higher level of preciseness. The basic application scopes of OCL were introduced by short examples and an overview of the concepts was given. A briefly summary of the history and similar languages rounds the introduction of OCL.

The next chapter focus the analogy between OCL and SQL. It contains a short explanation of SQL, an overview of model transformation patterns and an illustration of the relation between OCL and SQL.

3 On the Analogy of OCL and SQL

This chapter deals with the analogy of OCL and SQL and is divided in three parts. First, a short overview in the Structured Query Language (SQL) [**7**] and Relational Database Management Systems (RDMS) is given. This overview does not include a comprehensive description of SQL and RDMS. Therefore other works [**26**], [**27**] exist that illustrate an extensive introduction in the development of SQL, the foundational concepts and the distinction of the evolved SQL standards. As well as the platform specific implementations of several leading RDBMSs are discussed. However, the Section 3.1 summarizes the main ideas and historical relations in a generalized point of view and concentrates on the usage of SQL as query language. Second, the analogy between OCL invariants and SQL queries is illustrated by means of the UML motivating example in Figure 6 and the advantages and limitations are discussed. And third, related approaches using patterns for mapping OCL constraints to relational database integrity constraints are discussed in comparison to this work.

3.1 Overview of SQL and RDMS

In computer science there is a long-standing approach for modelling data information, based on Entity Relationship diagrams [**20**] as specification language, relational database for persistence issues and the SQL for querying the data.

In 1969 Edgar F. Codd proposed a research work about the development of a relational model for database systems. The approach follows a first-order predicate logic with the purpose of specifying and querying data in a declarative manner. Unlike network and hierarchical databases, the relational model consists of intuitive concepts for storing any type of data in a database and provides the base for relational database management systems. “An RDBMS is defined as a system whose users view data as a collection of tables related to each other through common data values” [**27**]. The related data is stored in tables that are composed of rows and columns. Tables cannot be considered in isolation, as there are usually relationships or associations between them. Such associations are expressed by unique, identifying columns of data, so called keys within a table. Nowadays, RDBMS is the predominant type of database systems, managing operations such as selections, projections and joins. The process for consistency in designing relational databases is known as normalization.

One of the major features of RDBMS is the support for the manipulation of data expressed by the Structured Query Language (SQL) [**7**]. SQL is an international standardized query language for accessing relational databases. The initial version, called SEQUEL (Structured English Query Language) was invented by IBM[7] in the 1974/1975 and was designed to manipu-

[7] http://www.ibm.com

late and retrieve data in relational database management systems of IBM. The first commercial implementation of SQL for database systems was introduced by Oracle[8] in 1979. Since its first incarnation, a number of different software companies recognized the potential of SQL and developed their own versions in response to specific professional needs. The demand of a standard was wanted throughout the leading RDBMS vendors. The syntactical base was within the implementations almost the same, but they were defined in specific SQL dialect. In addition, platform specific dependencies and operations have to be adapted. In 1992 the standardization organizations ANSI and ISO developed the so called SQL/2 or SQL-92 standard [**7**]. The Standard SQL is defined as a declarative query language, based on set operations of the relational algebra to retrieve data in relational database. Further evolvements and continuous improvements are described in [**27**], which are nowadays subject of the Joint Technical Committee ISO/IEC JTC. However, there are many extensions to the standard, which add procedural programming language, provided by the leading vendors. These incompatible extensions and different SQL implementations provided by the leading vendors are major points of criticism on the standard. Another criticism is the deviation of from its theoretical foundation, the relation model and its tuple calculus. While SQL provide a list of rows as result, the relational model delivers a set of tuples represented as table.

However, as a result, SQL became the most widely used database language in business and industry and the favourite query language for relational database management systems running on central or distributed systems. In the meantime, SQL is more than a pure query language, although the most common operation is the query. In the following section the concept of queries using SQL statements is described in details, because it follows the principle of OCL to produce no side-effects. The analogy between SQL queries and QCL invariants is shown by examples in further sections.

3.1.1 Querying relational database using SQL

A query is a method to retrieve data from any source, which allows the users to specify the data they want to enquire. Typically, it is used to access relational database management system by SQL statements. Data retrieval needs to be as easy as possible because the people, who write the queries, are not always those, who designed the database. Considerably, the language constructs for querying databases has to powerful enough to deal with all the user requirements in an intuitive manner.

A query expressed by the SELECT statement, represents a set of elements. The SELECT keyword allows the user to describe the desired data, in what order the data is arranged or what calculations are performed on the fetched data. Therefore the standard of SQL [**7**] provide up to six clauses, where the first two are mandatory. A simple SELECT state-

[8] http://www.oracle.com

ment only requires a table and a list of the desired columns within the table. The following example shows a query that returns a list of persons.

```
SELECT * FROM Person
```

The asterisk is used to refer all columns of the queried table. If only the *firstname* and the *lastname* of the persons in the table are needed, the asterisk is replaced as followed.

```
SELECT firstname, lastname FROM Person
```

In contrast to other SQL constructs, it is the most complex command due to the user requirements. Queries are responsible for planning, optimizing and performing the physical operations to generate the desired result as efficient as possible. Therefore the optional keywords and sub queries provide a powerful concept (see syntax for queries in [**7**]).

- The first mandatory part of a SELECT statement is an expression, which defines the desired columns in the result set. Therefore aggregate functions, renaming expressions or just a column name of a specified table in the FROM clause can be used.
- The FROM clause (mandatory) refers the necessary table(s) from which the data is retrieved. It can include different kind of joins or sub queries to specify the desired data and to optimize the query process.
- The WHERE clause is used to limit the number of affected rows of a query. It enables a restriction of the result set by defining criteria, which eliminates all rows that does not fulfil a given these criteria expressed by predicates.
- The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns or to eliminate duplicate rows.
- The HAVING clause was added to SQL because the WHERE keyword could not be used in conjunction with aggregate functions. It contains a predicate to filter the rows resulting from the GROUP BY clause.
- With the ORDER BY clause, the result set can be sorted by setting criteria. The sort criteria can be - but not limited to - column names, expressions, arithmetic operations or user-defined functions (optionally ascending or descending). The results of expressions are evaluated and are used to sort the result set.

The following example demonstrates the powerful constructs of SQL including all mentioned keywords based on the exemplary mapping of the motivation example to relational databases in Chapter 3.2.1. The SELECT statements return those managers and their salary that earn more money than the average income of all managers sorted by the highest income.

```
SELECT p1.pid, SUM(j1.salary)
FROM Person p1, Job j1, Company c1
WHERE p1.pid = j1.pid AND j1.cid = c1.cid AND p1.pid = c1.manager
GROUP BY p1.pid
HAVING SUM(j1.salary) > (SELECT AVG(salary) as average FROM Person p2,
                         Job j2, Company c2 WHERE p2.pid = j2.pid
                         AND j2.cid = c2.cid AND p2.pid = c2.manager)
ORDER BY SUM(j.salary) ASCENDING
```

The select expression includes the personal ID of the table *Person*, followed by the aggregate function SUM, which summarize the salary of each person. The FROM clause contains the necessary tables (*Person, Job, Company*) and use abbreviations (*p1, j1, c1*) for an easier referencing within the query. These references explain the relations between the tables to identify the managers and their salary, which is shown in the WHERE clause. The conditions in the WHERE clause could also be expressed using JOINS in the FROM clause. For the sake of demonstration, the conditions are defined in the WHERE statement as criteria to limit the number of elements in the result set. The GROUP BY clause group the result set to calculate the sum of each manager. The calculated sum is then compared with the average income of all managers, which is computed in a sub query. The HAVING clause filters the rows to identify only those manager that earn above the average income of all managers. Last, but not least the ORDER BY statement defines the sort criteria for the final result set.

A step-by-step introduction can be found in [**27**], which explains the details of the foundational concepts in a structured manner.

3.2 Analogy between OCL invariants and SQL queries

Before the similarity between OCL expressions and SQL SELECT statements can be shown, the object-relational impedance mismatch, which occurred in the 1990s, is discussed briefly. It reveals that the two paradigms are fundamentally different.

Object-oriented systems encapsulate data and behaviour in objects, whereas relational database systems store the data in tables. Objects are defined behind an interface and have a unique identity. On the other hand, relational databases are based on the mathematical concept of the relational algebra. A relational database is a self-descriptive repository of data that follows a defined structure or schema. These schemas stay relative static, while the data usually change very often.

To resolve the contradiction techniques for a direct object-relational mapping (ORM) that provide the ability to convert object-oriented data models in relational database schemas, was developed. Frameworks like Hibernate[9] provide comprehensive support for the design of database and programming code. An application can thus be developed in one and the same conceptual framework. Nowadays, the integration of relational databases into ob-

[9] http://www.hibernate.org/

ject-oriented applications is state of the art in software development. The disadvantage of the approach is that the strength and skills of RDBMS are not used, which leads to inefficient performance.

One of the easiest realizations is the commonly used class-to-table mapping [**28**], which is based on the UML class diagram technique to design the structure of components. It may be obtained by the means of the following steps:

- Entities or classes are translated into tables. Attributes of entities represent columns of the translated tables.
- While relations are mapped into a set of tables or foreign key constraints depending on their multiplicity.
- User-specific types are described with domains in the Data Definition Language (DDL) within the SQL standard.

The following sections leads through the mapping of the UML class diagram in Figure 6 to an equivalent relational database schema, using equivalent data for the object-oriented and relational models (see Chapter 3.2.2) and the mapping of OCL to SQL by means of examples.

3.2.1 Exemplary mapping of UML model to relational database

For the demonstration of the analogy of OCL invariants and SQL queries, the motivation example in Chapter 2.1.4 has to be mapped to the relational database schema. Therefore the class-to-table mapping in [**28**] was chosen. Each class in Figure 6 represent a table. Relationships are translated into a set of tables or foreign key constraints depending on their multiplicity. According to the class-to-table approach, the following tables are determined:

```
create table PERSON (       PID integer PRIMARY KEY,
                            FIRSTNAME varchar not null,
                            LASTNAME varchar not null,
                            AGE integer,
                            BIRTHDATE date not null,
                            GENDER Gender not null,
                            UNEMPLOYED Boolean not null,
                            MARRIED Boolean not null,
                            WIFE_HUSBAND integer references PERSON,
                            PARENT_CHILDREN integer references PERSON)

create table COMPANY (      COID integer primary key,
                            NAME varchar not null,
                            ADDRESS varchar not null,
                            NUMBEROFEMPLOYEES integer,
                            MANAGER integer references PERSON)

create table VEHICLE (      VID integer primary key,
                            REGISTRATION varchar not null,
                            VEHICLE_OWNER integer references PERSON)
```

```
create table CAR (          VID integer primary key,
                            CATEGORY CarCategory not null,
                            foreign key (VID) references PERSON)

create table MOTORCYCLE (   VID integer primary key,
                            foreign key (VID) references PERSON)

create table ACCOUNT (      AID integer primary key,
                            ACCOUNTID varchar not null,
                            BALANCE float not null,
                            ACCOUNT_OWNER integer references PERSON)

create table JOB (          PID integer references PERSON,
                            COID integer references COMPANY,
                            STARTDATE date not null,
                            ENDDATE date,
                            SALARY float,
                            primary key(PID, COID))

create domain SEXTYPE character check (value in 'm', 'f')
create domain CARCATEGORY character check (value in 'l, 'm', 'c')
```

3.2.2 Exemplary data for the UML model and the relational database

In addition to explain the analogy between OCL and SQL, instances of the objects for the UML class diagram and equivalent data stored in the relational database is needed. The corresponding data entries in the database are listed in the following tables. The instances of the UML class diagram are illustrated as an object diagram (see Figure 11).

PID	First-name	Last-name	Age	Birthdate	Gender	Unem-ployed	Married	Wife_ Husband	Parent_ Children
P1	Markus	Siedler	17	24.03.1996	m	True	True	Null	Null
P2	Karin	Popp	54	22.12.1959	f	False	True	Null	Null
P3	Herbert	Humer	25	15.10.1988	m	False	True	Null	Null
P4	Michael	Eder	22	02.05.1991	m	False	False	Null	Null

Table 3 - Data entries in table Person

VID	Registration	Vehicle_Owner
M1	Missing	P3
M2	Done	P1
C1	Done	P3

Table 4 - Data entries in table Vehicle

For the demonstration the special types for vehicles are not used, because they only include the foreign key reference to its super type.

COID	Name	Address	NumberOfEmployees	Manager
CO1	SP	Nestroyplatz 1, 1020 Vienna	25	P2
CO2	OS	Steingasse 3, 1030 Vienna	87	Null

Table 5 - Data entries in table Company

PID	COID	StartDate	EndDate	Salary
P4	CO1	02.07.2009	Null	1100
P4	CO2	05.01.2011	Null	450

Table 6 - Data entries in table Job

The identifiers in the relational database tables and the instances within the object diagram are extended by the first letters of the entity. For example, the data entries for Person are using a *P* in front of the identifier. This modification is done to provide an easier comparison between the models and a better understanding for following OCL2SQL examples.

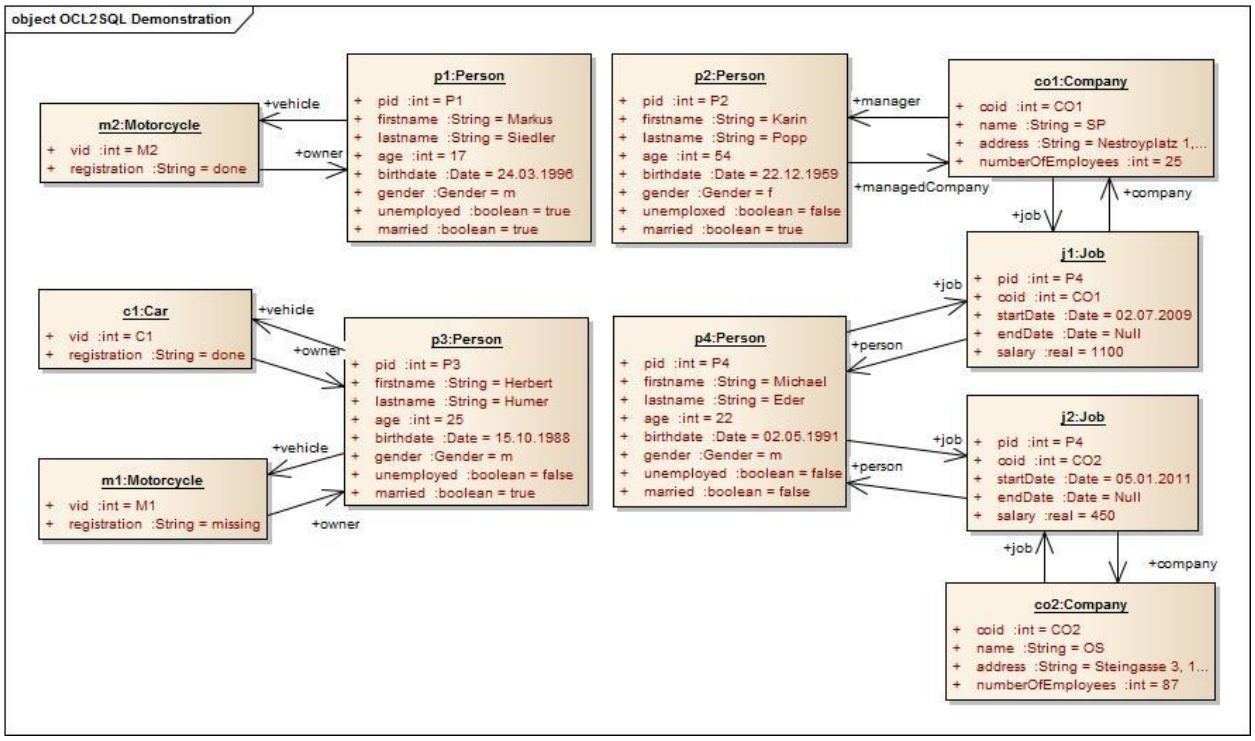

Figure 11 - Object diagram for OCL2SQL demonstration

The data entries describe persons with individual properties and relations. The persons P1 and P3 own one or two vehicles. On the other hand, the persons P2 and P4 have relations to companies. P2 is the manager of the company CO1, where P4 is just an employee of CO1, described with the object J1. The job J1 represents the relation of an employee and extends it with properties like the salary that the person earns. Person P4 is very diligent – P4 has a second job at the company CO2. If the monthly income is calculated the salaries are summed up as we show in one of the following OCL/SQL examples in the next section.

3.2.3 Exemplary mapping from OCL to SQL

This section deals with the query languages of OCL and SQL by means of examples. Each demonstration is structured as followed: First, the OCL invariant is defined on the UML class diagram in Figure 6, by setting the context of the OCL expression. Comments describe the meaning of the expression in natural language. Second, an equivalent SQL query based on the relational database schema in Chapter 3.2.1 is specified to depict the analogy between the query languages and to highlight the result sets. The SQL statements use sub queries for an easier understanding of the relational algebra. A sub query in the following examples always represents all data entries of a type (see Mapping Example 2). For the implementation of the ADOCLE, we could figure out transformation patterns between OCL and SQL. Finally, the output in form of a result set follows based on the exemplary data in Chapter 3.2.2.

Mapping Example 1 - Attribute

OCL:
```
-- Transformation pattern for attributes
-- Full-aged married persons.
context Person inv fullAgedMarriedPersons:
self.age > 18 and self.married = true
```

SQL:
```
SELECT * FROM Person as p WHERE p.age > 18 and p.married = true
```

Result set:
```
p2, p3
```

Mapping Example 1 shows the transformation pattern for attributes. Attributes in UML accessing with OCL invariants can be referred by SQL in a similar way. Therefore the predicates (*self.age > 18, self.married = true*) in OCL has to be mapped in predicates that are placed in the WHERE clause. The next cases explain the different kinds of associations and its transformation pattern. The OCL navigation *self.manager* in Mapping Example 2 represents all managers. Therefore all instances of persons have to be checked, if they have the relation *manager* to an instance of company. As output, the instances of persons are required, that is done by the first line in the SQL query.

Mapping Example 2 - Many-to-One Relation Manager

OCL:
```
-- Transformation pattern for associations
-- Many-to-one relation
-- Manager of a company
context Company inv: getTheManagerOfACompany
self.manager
```

SQL:
```
SELECT p.* FROM (
      SELECT c.* FROM Company c,  -- all instances of companies
      SELECT p.* FROM Person p)   -- all instances of persons
      WHERE c.manager = p.pid     -- relation manager
```

Result set:
```
p2
```

Mapping Example 3 - Many-to-One Relation Vehicle Owner

OCL:
```
-- Owner of a vehicle
-- Many-to-one relation
context Vehicle inv: getTheOwnerOfAVehicle
self.owner
```

SQL:
```
SELECT p.* FROM (
      SELECT v.* FROM Vehicle v,
      SELECT p.* FROM Person p)
      WHERE v.vehicle_owner = p.pid
```

Result set: p1, p2

The Mapping Example 3 shows the same behaviour as Mapping Example 2. It just describes another many-to-one OCL navigation in the UML class diagram, whereas the Mapping Example 4 depicts the other direction of the relation. The one-to-many relation *self.vehicles* represents all vehicles of all persons. The SQL statements of Mapping Example 3 and Mapping Example 4 look very similar. The used join between the tables is the same, but the queried result set is different. In the Mapping Example 3, the persons, who already have a vehicle, are listed. On the other hand, the Mapping Example 4 shows a result set of vehicles that are related to a person.

Mapping Example 4 - One-to-Many Relation

OCL:
```
-- Navigation from Company to Person
-- One-to-many relation
context Person inv: ownedVehicles
self.vehicles
```

SQL:
```
SELECT v.* FROM (
      SELECT p.* FROM Person p,
      SELECT v.* FROM Vehicle v)
      WHERE p.pid = v.vehicle_owner
```

Result set: m1, m2, c1

Mapping Example 5 - Combination of relations and attributes

OCL:
```
-- Navigation from Company to Person
-- One-to-many relation
-- restriction to persion with id 1
context Person inv: ownedVehicles
self.vehicles and self.pid = 1
```

SQL:
```
SELECT v.* FROM (
      SELECT p.* FROM Person p,
      SELECT v.* FROM Vehicle v)
      WHERE p.pid = v.vehicle_owner AND p.pid = 1
```

Result set: m2

If the query of Mapping Example 4 is extended by a restriction to a specific person (PID is 1) – appending an attribute mapping, the result set is limited to the following entities (see Mapping Example 5).

Mapping Example 6 - Operation

OCL:
```
-- Transformation pattern for operation
-- Calculate the salary of a person
context Person inv: salary of a person
self.job.salary->sum()
```

SQL:
```
SELECT SUM(j.salary) FROM (
      SELECT p.* FROM Person p,
      SELECT j.* FROM Job j)
      WHERE p.pid = j.pid
      GROUP BY p.pid
```

Result: `1550`

OCL Operations can be described with aggregate functions of SQL. There are several transformation patterns depending on the operation in OCL. For the Mapping Example 6, a grouping for each person in conjunction with the aggregate function *SUM* is necessary. The patterns are founded on a comprehensive study of possible transformations of OCL expressions to SQL [**29**]. The author describes the patterns depending on the type hierarchy of OCL and categorizes them.

3.3 OCL2SQL transformation approaches

There are several works about transformation approaches, especially for mapping object-oriented models to relational database schemas. Most works concentrate on the transformation of UML class models to the Data Definition Language (DDL) of the SQL-92 standard. A commonly used solution, the class-to-table mapping is explained in the previous sections. The purpose of such approaches is the integration of relational database systems in object-oriented software systems. In the context of the UML four-layer metamodel hierarchy, such database application models are defined on layer M1 and the instances are placed in layer M0 (see Chapter 2.1.1).

Based on this concept, the authors of the paper [**30**] reported on a systematic study of the use of OCL expressions in the context of relational databases. They developed an approach for translating OCL expressions in SQL views. The purpose of [**30**] is to extend the UML2SQL transformation engines with the powerful OCL expressions. Therefore, OCL mapping patterns are realised by equivalent SQL queries with the VIEW approach [**31**]. Database enforced integrity constraints such as CHECK only refer on tuples of one table. Indeed typical OCL expressions navigate through more entities, and so the OCL expression has to be mapped to multiple tables respectively relations. According to [**30**], SQL views support the

requirements for translating OCL invariants to SQL queries using transformation patterns. An integration of view-based integrity check on the persistence layer of the database can be done with database-specific triggers. In addition, a template-based engine for code generation supports vendor-specific SQL dialects too. Such an approach could also be used for validating the instances against the model. In the context of this master thesis, the validation of models against its metamodel and the WFRs can be managed.

The paper [**32**] presents an integrated approach for the development of enterprise information systems (EIS). The idea concentrate on a conceptual metamodel that describe several aspects of EIS software: application functions, business rules and the database schema. In contrast to the approach in paper [29], the structural aspects of EIS, such as business concepts, instances, relations and static constraints are defined by means of the Entity-Relationship (ER). The database schema is generated by an ER-to-SQL mapping algorithm, which acts similar to the class-to-table approach. For action or business rules, assertions and derivation rules OCL expressions are used. The combination of the expressive power of the ER conceptual metamodel and the dynamic OCL expressions allow the framework, the specification of structural and behavioural aspects. The framework written in Java, translate OCL expressions in stored procedures expressed by SQL. Also basic CRUD (Create, Read, Update and Delete) operations on entities are translated in stored procedures.

Other well-known approaches, like the Java Persistence API (JPA)[10] in conjunction with Hibernate[11], provide an automatically generation of tables using annotations from a conceptual model. Nevertheless, there is no support to manipulate the conceptual entities.

AndroMDA[12] uses another way for translating UML/OCL to SQL. It is an open source Model-Driven-Architecture framework that can take UML models and generate code for other frameworks like Hibernate. In the case of Hibernate the OCL expressions are translated in HQL, the logical query language of Hibernate. The created query is than translated in the database-specific language using the SQL Dialects of Hibernate. Problems for supporting database-specific issues are outsourced by using the Hibernate framework.

[10] http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[11] http://www.hibernate.org/

[12] http://www.andromda.org

However, the main purpose of the mentioned works or frameworks is the generation of SQL statements based on a source metamodel. Depending on a source metamodel, the target metamodel is translated using transformation patterns, for example UML/OCL to SQL or other programming languages. Figure 12 depicts two models, a class diagram and a physical data model, which follows out of the class-to-table mapping approach mentioned in [**28**]. The metamodel is semantically equivalent. Thus, the instances of Person of the class model are equivalent to the instances in the physical data model.

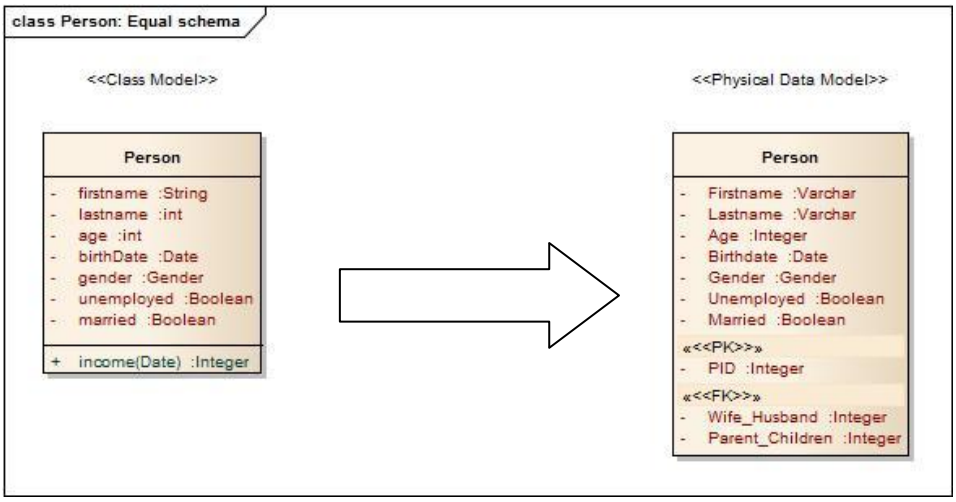

Figure 12 - Equal schemas due to the class-to-table mapping approach

The aim of such approaches is the integration of relational database systems in object-oriented software systems. In the context of the problem statement of this master thesis, we focus on an adaptable, reusable approach for transforming OCL expressions in other environments or CASE tools. The OCL expressions are applied to a selectable source metamodel, which is mapped to a target metamodel of the user's choice. Hence, the schemas are already defined and are not certainly equivalent, but overlap semantically.

For example, the class model in Figure 13 contains two attributes to identify the firstname and the lastname of a Person – instance of class model (Person {firstname='Christoph', lastname='Zehetner'}). In the physical database schema these information is distributed over two instances (Person {name='Christoph'}, Family {name='Zehetner'}). Both schemas may express semantically equivalent information, but the schemas are not equivalent at all. However, not all concept of the class model are provided by the concepts of the physical data model. For instance, the attributes *gender, unemployed* and *married* in the class model in Figure 13 are not supported in the physical data model. Therefore, the concepts of the class model have to be mapped to the concepts of the physical data model.

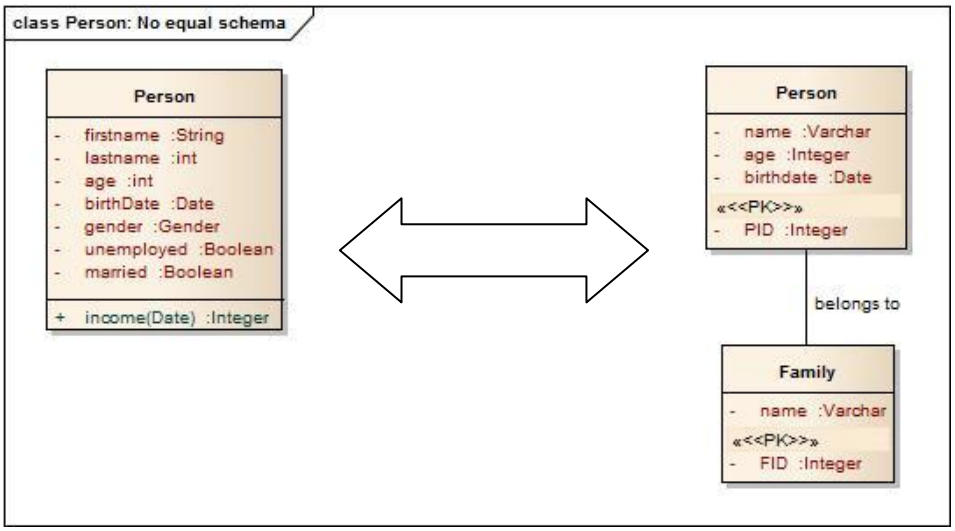

Figure 13 - Semantically overlapping schemas

In general a metamodel mapping between the source and target metamodel is necessary; to identify the semantically equivalent corresponds of the target metamodel. The big difference between the previous mentioned works is that the target metamodel is generated out of the source metamodel, whereas ADOCLE needs the metamodel mapping to support an automatic validation. Nevertheless, the mentioned solutions provide a foundation to integrate a metamodel mapping in an independent OCL2SQL transformation approach, which follows in the next chapter.

4 Realizing ADOCLE

The chapter deals with the prototypical implementation of an adaptable approach, ADOCLE for transforming OCL expressions in other environments. First, we give an overview of the transformation process of OCL expressions applied to a selectable source metamodel. Second, the architectural approach and its main components as well as the design decisions are explained. Furthermore, the user interface of the prototype is illustrated. Finally, the chosen development method and environment, as well as the used platform and environment tools are documented.

4.1 Overview of the ADOCLE

The aim of the prototype is the transformation of OCL expressions in other environments. Therefore, we defined a process, which illustrates the information flow and the relations between the main components during the transformation. Each component manages its tasks in an independent module. This allows a reuse of these modules. Figure 14 shows an abstract overview of the transformation process, containing the main components, their dependencies as well as input and output objects.

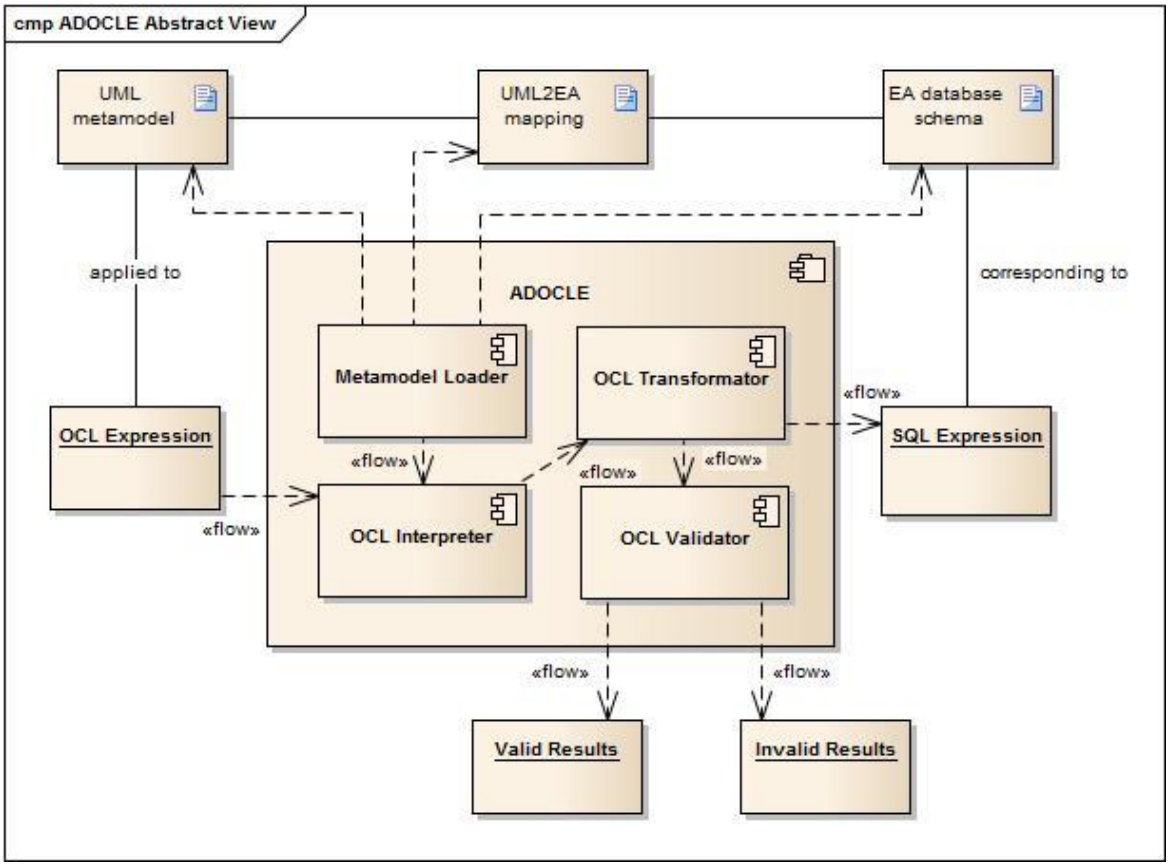

Figure 14 - Abstract view of ADOCLE

For the prototype we use the UML metamodel as source metamodel and the well-formedness OCL expressions in the UML specification as OCL input. Enterprise Architect is chosen as a UML modelling tool. Because, EA uses a relational database to store the model information, we decide to use the physical database schema of the EA as target metamodel. EA's database schema is described with Structured Query Language (SQL) and based on the SQL-92 standard. The mapping between the UML metamodel and the EA database schema is based on patterns, which identifies the semantically equivalent UML metamodel elements that are stored in the database of EA. When other metamodels are chosen, the metamodel mapping has to be defined by patterns depending on the chosen metamodels. The transformation process is the same, but the transformation rules have to be adapted according to the chosen metamodels. Consequently, the mentioned artefacts (UML metamodel, EA database schema and the metamodel mapping between them) provide the base for the transformation process, which are loaded at the beginning by the *Metamodel Loader*. For an easier understanding, Figure 15 depicts the transformation process from the perspective of an ADOCLE user. The *Interactive OCL Console* describes components of the user interface, which are explained in Chapter 4.3 in detail.

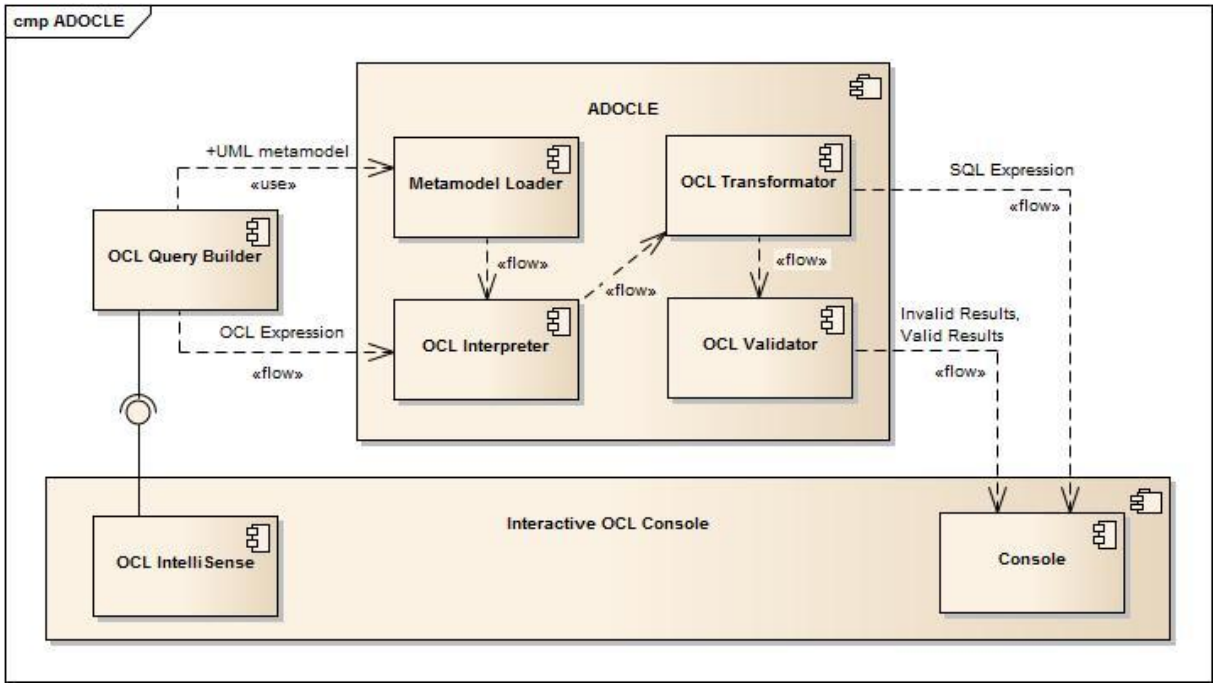

Figure 15 - ADOCLE

The OCL Query Builder provides the ability to read the loaded source metamodel (UML metamodel) and returns the next possible text segments. Hence, the defined OCL expression is always applied to the source metamodel. These text segments can be attributes, association ends of the source metamodel or operations depending on the current position in the metamodel (see Chapter 2.2.3). The main task of the ADOCLE prototype is to analyse the

OCL expression and generate an equivalent SQL expression for the target metamodel – the database schema of EA. Therefore, the OCL interpreter analyse the given text segments (OCL expression) and generate an abstract syntax tree (AST), which checks the conformance of the OCL expression against the OCL grammar in [**3**]. As next step, the OCL expression in form of the AST is translated to an SQL expression. The OCL transformator navigates through the AST and identifies implemented pattern and resolve the metamodel mapping depending on the parts within the given AST. Finally, the generated SQL expression is executed to provide an automatic validation of the well-formedness rules within the EA database. The core components and the architecture of the ADOCLE are explained in the following sections in detail.

A user interface supports the ADOCLE user by the generation of the OCL expression with a flexible OCL Query Builder using IntelliSense. The results of the created SQL expression are listed. Typically an OCL expression results in a positive state and delivers the valid elements. But the more interesting elements are those, which do not fulfil an OCL expression. Therefore, we inverse the OCL input to return those elements that are not conform to the given OCL expression. Chapter 4.3 provides further details how to work with the user interface.

4.2 Architecture of the ADOCLE

This chapter deals with the architecture and the implementation of the ADOCLE in detail. The architecture approach is module-based. It provides an easy integration of metamodels using templates for parsing or loading source metamodels, target metamodels and the mappings between them. An adaptable transformation unit allows defining mapping rules between source and target metamodels, which is used to generate an equivalent expression for the target metamodel from an OCL expression applied to the source metamodel. In the case of the Enterprise Architect, the target metamodel is the database schema of the tool that leads to an OCL2SQL transformation. Metamodel independent components can be better reused for other metamodel transformations. The architecture is designed for openness and modularity.

4.2.1 Metamodel loader

The first module is responsible for parsing and loading metamodels, which are described with abstract syntax trees (AST, regardless of a source or target metamodel). The paper [**33**] addresses theoretical and practical aspects of implementing multi-stage languages using abstract syntax trees and illustrate why this strategy can be particularly useful for implementing domain-specific languages in a typed, functional setting. Multi-stage languages using ASTs allow computations in every stage (every part of the AST) to provide the ability of executing actions or to refer to values from previous stages. In the paper significant gains (in

the typed functional setting) are demonstrated, if the implemented strategy is used in conjunction with runtime source-to-source transformations.

In the case of ADOCLE, a general interface is defined to provide a metamodel independent parsing and loading, which are expressed by abstract syntax trees. The general interface describes the data structure of a metamodel environment (see Figure 16) to compare different metamodel elements and to define mapping rules between them. The idea is to express any kind of metamodel by an abstract syntax tree using a self-defined structure – metamodel environment. For example, the UML metamodel uses classes, properties and associations to describe the main components and relationships. The EA metamodel uses a relational database schema expressed by SQL that is structured with tables, fields and joins instead. Hence, the semantic information is prepared in a similar manner but differs structural in different schemas. The mapping between the UML metamodel and the EA schema deals with structural differences to link the semantic equivalent information in both schemas. However, one UML metamodel element is often expressed by a combination of different parts of the EA schema. Each metamodel is parsed and mapped to the self-defined data structure (environment), which is represented by the abstract syntax trees. Afterwards these trees are serialized for later reuse. This will reduce the initialization time of ADOCLE. The serialized ASTs constitute a complete data source for other modules.

The metamodel environment parser is responsible for searching elements in an environment that represents a metamodel in the self-defined data structure. Therefore a depth-first search (DFS) algorithm [**34**] is applied for traversing the tree structure. The algorithm starts at the root and explores as far as possible along each branch before backtracking. This algorithm was chosen, because it is easy to implement and adequate for the prototype development. If the idea of ADOCLE will achieve product maturity, more efficient algorithms should be taken into account.

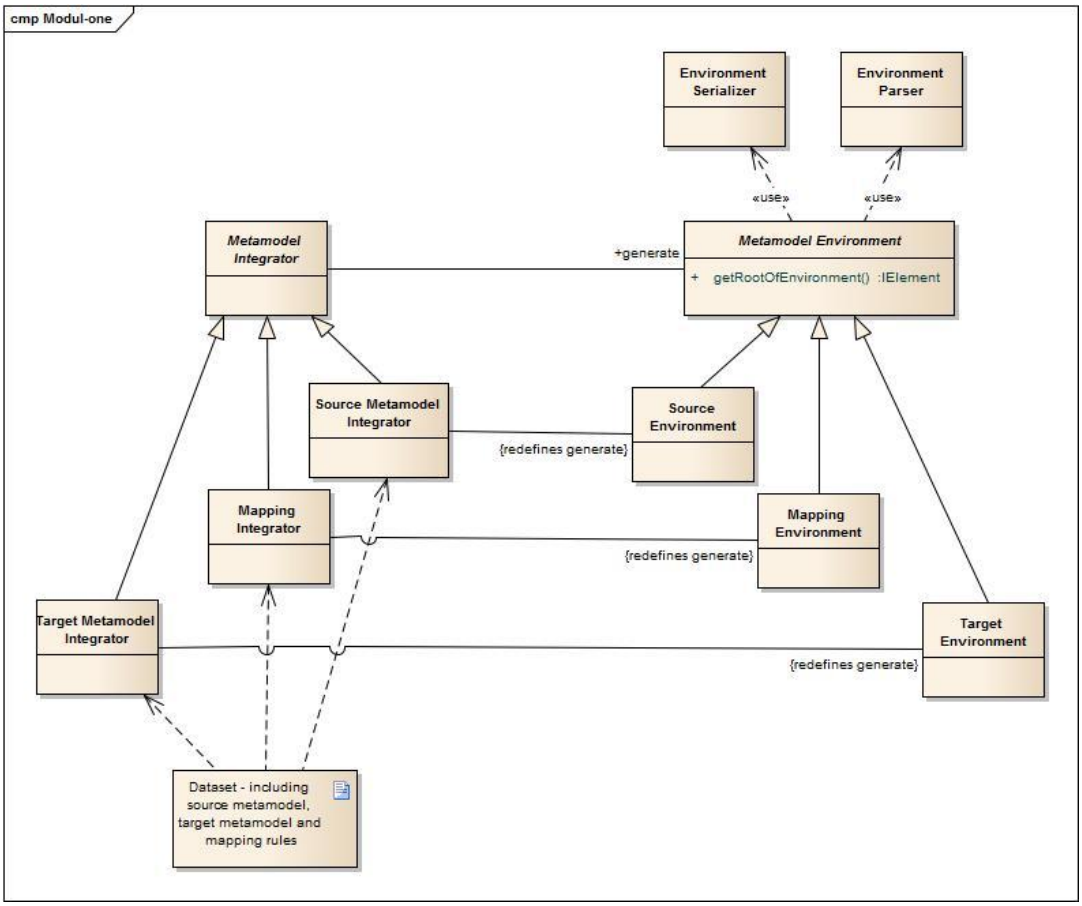

Figure 16 – Metamodel Loader

The benefit of the mapping to a self-defined structure is that transformation rules can be easier defined between any mapped metamodel due to the consistent structure (see Chapter 4.2.1.1). The mapping to a self-defined structure is a template-based approach. The prototype realizes the parsing of XML-based metamodels. Other file formats are also possible, but there are no templates already available.

But the main challenge of the first module is the determination of the schema mapping between the source and the target metamodel/schema. Hence, in this chapter the terms schema and metamodel as well as model and instances are used as synonyms, respectively. Terms like schema mapping or schema matching are often used interchangeably. In general, schema transformation is the translation of one (or more) schema in another (or more) or the combination of more schemas to a new schema. A transformation of a schema to another is called schema mapping, while the automatic detection of such mappings is named schema matching. A schema mapping consists of correspondences or associations that relate semantically equivalent elements of two or more schemas. In the case of ADOCLE, the schema mapping defines all semantically equivalent correspondences between the UML metamodel and the physical database schema of EA. These relations are used to derive transformation patterns to translate data from one schema as completely as possible into

another (see Chapter 4.2.1.1). There are two options to detect semantically equivalent correspondences between schemas: manually or automatically.

Manual schema transformation is the process of identifying semantic mappings, or correspondences between two or more schemas using no matching solution. Just humans make the decisions about the semantically relations between the schemas. On the other side, the automatically schema transformation use some kind of matching algorithm for the identification. The origins of schema matching lies in the area of database engineering in the 1980s and 1990s, as autonomous database started to federate [**35**]. Thus, the originally purpose of schema matching is either to merge two or more databases, or to enable queries on multiple, heterogeneous database. But schema matching becomes a critical problem in many application domains, such as data warehousing, e-commerce, semantic web, schema/ontology integration, etc. In current implementations, schema transformation is typically performed manually, which has significant limitations [**36**], or in best cases semi-automatically. Manual schema transformation is a tedious and time-consuming task, but it is more exactly in most cases. On the other side, automated schema matching provides a more comfortable way of finding semantically equivalent correspondences, but without a guarantee of correctness. Many diverse approaches to the schema matching problem have been proposed; while surveys [**36**], [**37**], [**38**] presented and compared the major contributions of the last decades. These works describe the problem of schema matching in detail and illustrate the different kind of heterogeneities and resolutions of them. In addition, classifications for the research work are developed and analysed to identify the directions in which research on schema matching is headed. Finally, the surveys discuss the advantages and disadvantages as well as the orientation for future research work. However, none of the schema matching methods in [**36**] – latest aforementioned survey - have reached a stage of being completely automatic. Some automatically algorithms described in [**36**] suggests potential matches, but humans often make the final judgement. Some researches, mentioned in [**36**], do not foresee fully automatic schema matching as a possibility, and orient their researches towards assisting human-performed schema matching.

According to the idea of ADOCLE, the major issue is the changeability of the source and the target metamodel. A metamodel contains the meta information of the stored data, for example, the instances. It could be designed as a tree in XML like the UML metamodel (see [**1**], [**2**]), as schema in a relational database (see [**5**]) or as an object structure in any programming language provided by an API or web service. The schema transformation has to be supported for all combinations due to the adaptive approach of ADOCLE.

Using an automated schema matching approach, most of the conflicts and heterogeneities (syntactical, structural, representational or semantic heterogeneity), described in the aforementioned surveys, have to be solved, which is quite difficult and time-intensive. Nevertheless, there is no guarantee that all matches are found and refer the semantically equivalent correspondences, which is essentially for ADOCLE.

We decided to use a manual schema transformation for the implementation of the prototype. This master thesis is more focussed on the functionality of the ADOCLE approach and not on solving the matching problem. For future work, a semi-automatically solution for the schema matching could assist during the implementation of other adapter and reduce the specification of the metamodel mapping.

4.2.1.1 Manual Metamodel Mapping

The focus of the implementation of the manual schema transformation is led to keep it as simple as possible, because it provides the foundation for further environments. We use a solution that combines all information (source metamodel, target metamodel, mapping between them) in a single dataset. Therefore, the source and target schema was imported using a UML modelling tool. Usually, the Enterprise Architect was consulted.

The advantage of this approach is that each element of the source and the target metamodel get a global identifier within EA. Each metamodel has to be mapped to the self-defined structure. Possible name matching problems while parsing the metamodel are eliminated through the global identifier. In addition, searching in the self-defined structure becomes more efficient. Furthermore, the full functionality of the Enterprise Architect can be used like the visualization of the source and the target metamodel. The mapping rules between elements could be easily defined by drawing different kind of relations between a source metamodel element and a target metamodel element (see Chapter 4.2.1.2). Before the mapping rules can be determined, we defined a metamodel mapping concept (see Figure 17). It specifies the structure of the mapping patterns which is necessary to recognize the different kinds of defined mappings during the transformation process (see Chapter 4.2.3).

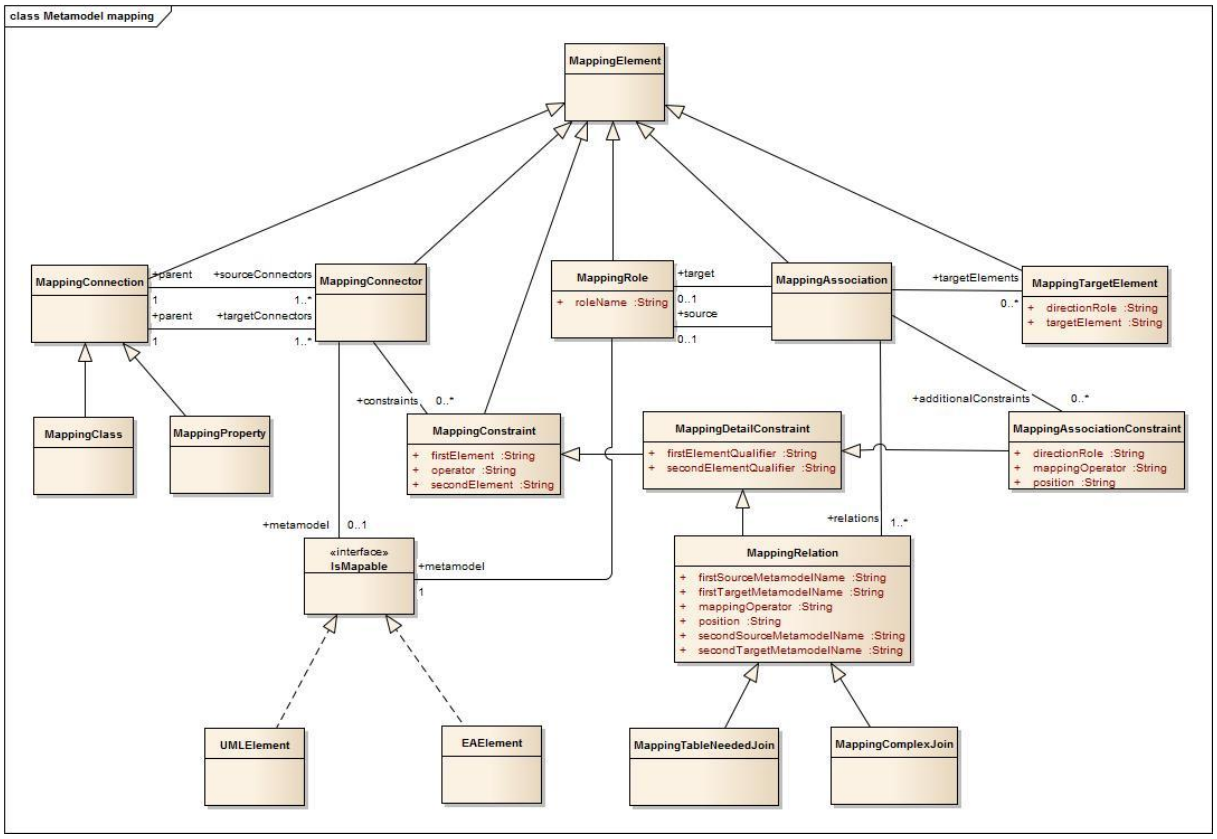

Figure 17 - Metamodel mapping concept

4.2.1.2 Metamodel Mapping Patterns

This chapter deals with the different kinds of metamodel mapping patterns that are used to define the manual schema mapping based on the metamodel mapping concept illustrated in Figure 17.

4.2.1.2.1 Element Mapping

The mapping of elements is defined by a direct relation between the elements that describe semantically equivalent correspondences in both metamodels. The connectors contain the details in form of constraints how the source metamodel element is mapped to the target metamodel element. The relation between the metamodels may be many to many (m:n). In most cases, the mappings are 1:n associations from UML to the EA schema. Figure 18 shows the mapping of the UML concept *State* to the corresponding parts in the EA schema. In the database schema of the Enterprise Architect a *State* is stored in the table *t_object* with the *Object_Type* ‘State’ and the *NType* ‘0’ or ‘8’.

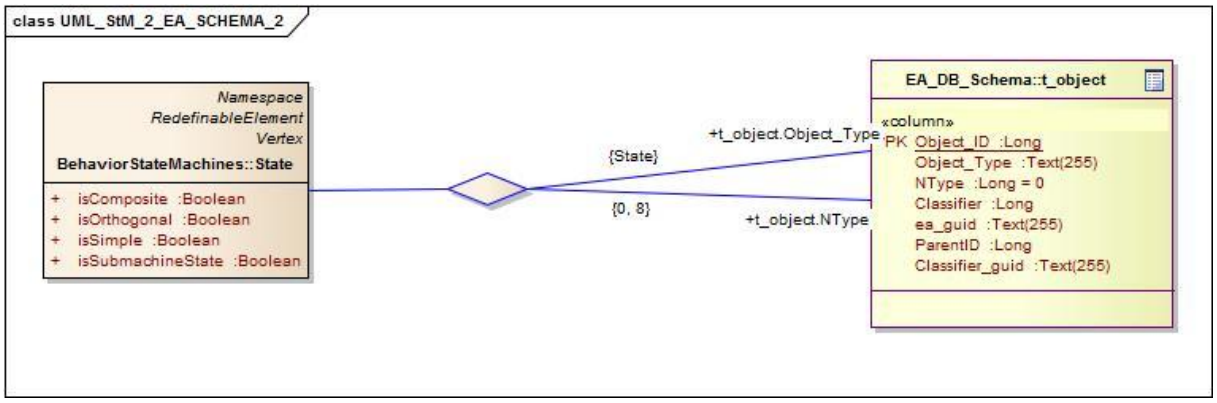

Figure 18 - Element mapping

4.2.1.2.2 Property Mapping

The mapping pattern for properties is very similar to the element mapping pattern. The relation refers to the property of the source metamodel and how it is described in the target schema. In Figure 19 the mapping of two properties of the UML concept *State* is depicted. It shows that the property *isComposite* is described in another target metamodel element than the property *isOrthogonal*.

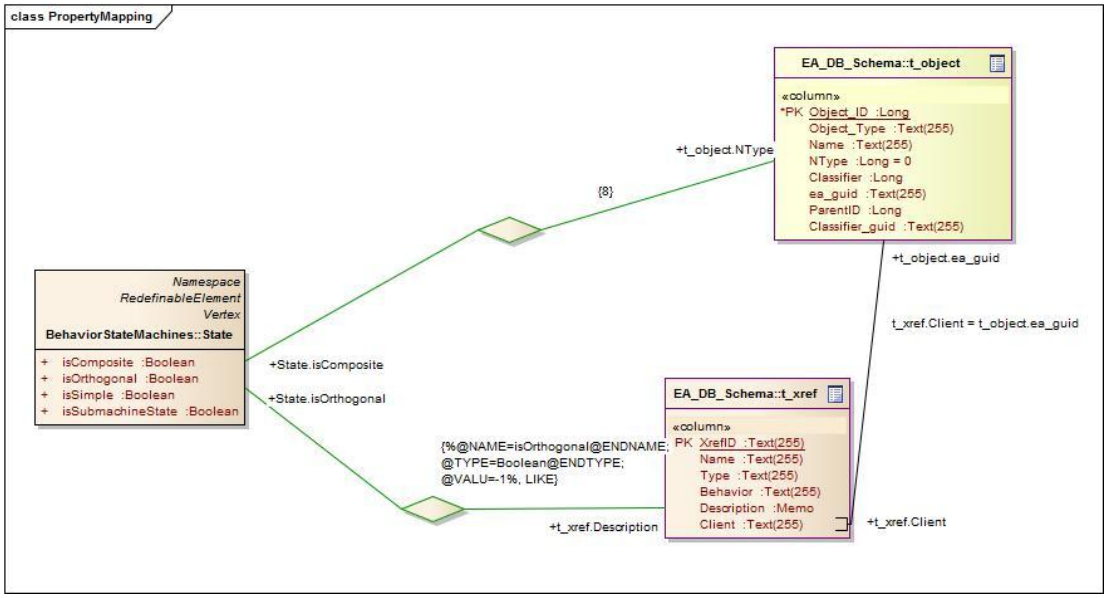

Figure 19 - Property mapping

4.2.1.2.3 Enumeration Mapping

An enumeration is a special kind of property. Therefore, a separate mapping pattern is necessary. Each enumeration literal of the source metamodel has to be defined to refer the semantically equivalent element in the target metamodel. Figure 20 depicts the mapping of UML concept *Pseudostate*.

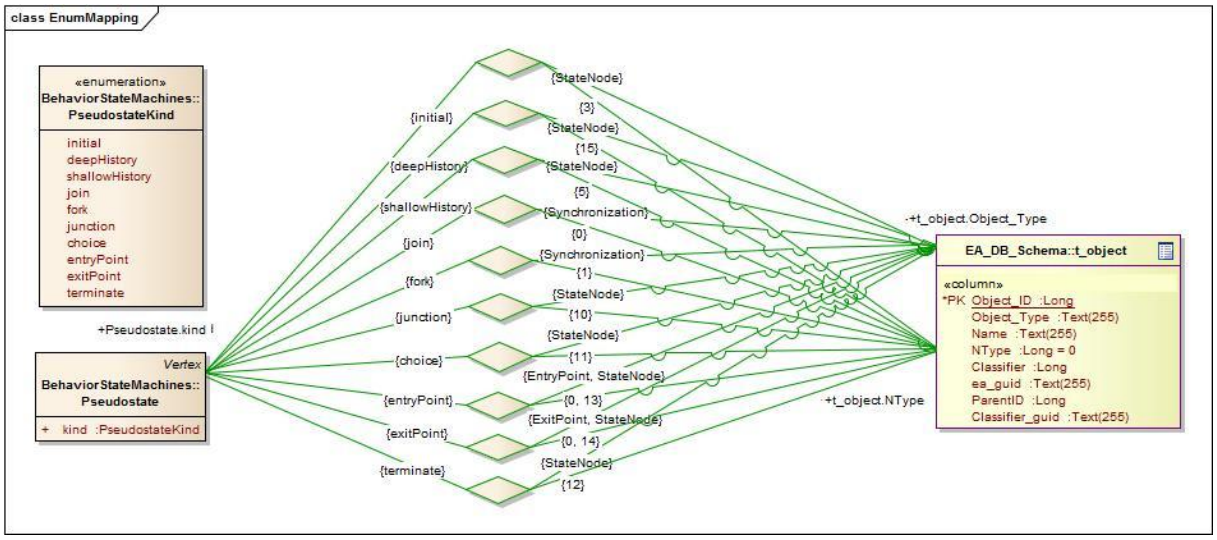

Figure 20 - Enumeration mapping

4.2.1.2.4 Association Mapping

The mapping of an association depends on the related elements in the source metamodel and the meaning of the association. The mapping is defined by constraints that represent a semantically equivalent association in the target metamodel. In Figure 21 the association between the UML concepts *State* and *Pseudostate* is depicted. In the case of the Enterprise Architect database schema, a join between two sets has to be defined, which is declared by the INNER JOIN constraint shown in Figure 21. The INNER JOIN constraint specifies the relation in the target metamodel using the required fields. In the following example, the association describes that a state can have connection points. A well-formedness rule of UML specifies that only pseudostates (UML concept *Pseudostate*) with a kind of *exitPoint* and *entryPoint* are connection points (UML concept *ConnectionPoint*). Additional constraints can be defined by using the keyword CONSTRAINT to explain the semantically equivalent representation of an association.

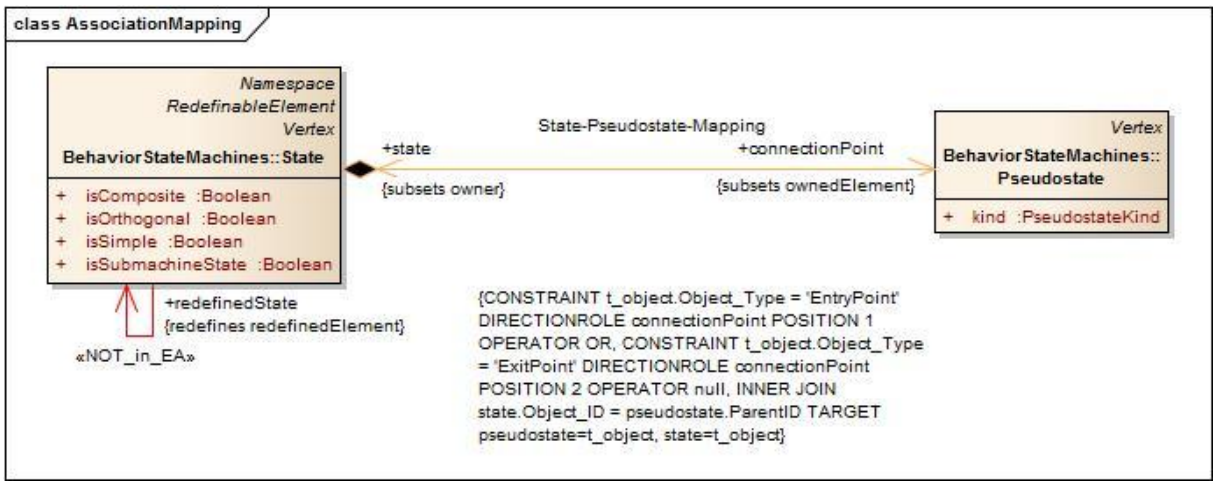

Figure 21 - Association mapping

4.2.2 OCL Interpreter

The second module is responsible for the resolution of OCL commands. The module takes a text segment as input and produces a valid homogenous object tree of the OCL grammar [**3**]. The OCL interpreter should be independent of other parts and easily maintainable, because the OCL grammar can change through updates of the specifications developed by the OMG.

A parser generator is used to generate automatically an OCL parser based on the OCL grammar. A parser generator is kind of compiler-compiler or compiler-generator. In 1963 Tony Booker has written the first compiler-compiler for the Ferranti Atlas computer at the University of Manchester and defined it as "an early compiler generator for the Atlas, with its own distinctive input language" [**39**]. According to Jørgensen's work [**40**], compiler-generators[13] are often emphasized as being the most important application of partial evaluation. The operational area is small formal languages. "Partial evaluation is described as a source-to-source program transformation technique for specializing programs with respect to parts of their input" [**41**]. Much partial evaluation work has concerned on automatic compiler generation from an interpretive definition of a programming language, but it also has important applications to metaprogramming.

The core of the module is generated by the ANTLR Parser Generator [**42**]. ANTLR[14] takes a modified version of the formal description of the OCL grammar (see Chapter 4.2.2.1), e.g. in Backus-Naur Form (BNF) [**13**], [**43**] or Extended-Backus-Naur Form (EBNF) [**13**], [**43**] and outputs a source code in a programming language of your choice. The source code of the module is separated in three parts: **a lexer, a parser and a treeparser.**

The lexer takes a stream of characters and emits a stream of tokens that are specified through the OCL Grammar [**3**]. The text segments of an OCL expression can be produced using the OCL Query Builder, which navigates through the source metamodel and provide possible text phrases. However, it is also possible to copy and paste an OCL expression, the OCL parser will check the correctness of the provided OCL expression. Characters like whitespaces can be flagged as unnecessary. The parser reads the token stream, typically the emitted stream of the lexer and generates an abstract syntax tree (see the structure of the AST in Figure 26 in Chapter 4.2.3.1). An AST is a purely abstract representation of the syntax, where a direct association between the production rules in the specification and the nodes in the tree is specified. The treeparser uses the AST as source to produce a homogenous object tree that reflects the OCL command. The object tree forms a consistent base for transformation rules, so that every element in the object tree can trigger actions during the transformation process.

---

[13] "A compiler generator is a program (or system) that given some machine readable formal description of a programming language produces a compiler for that language." [**40**]

[14] "ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages." http://www.antlr.org/

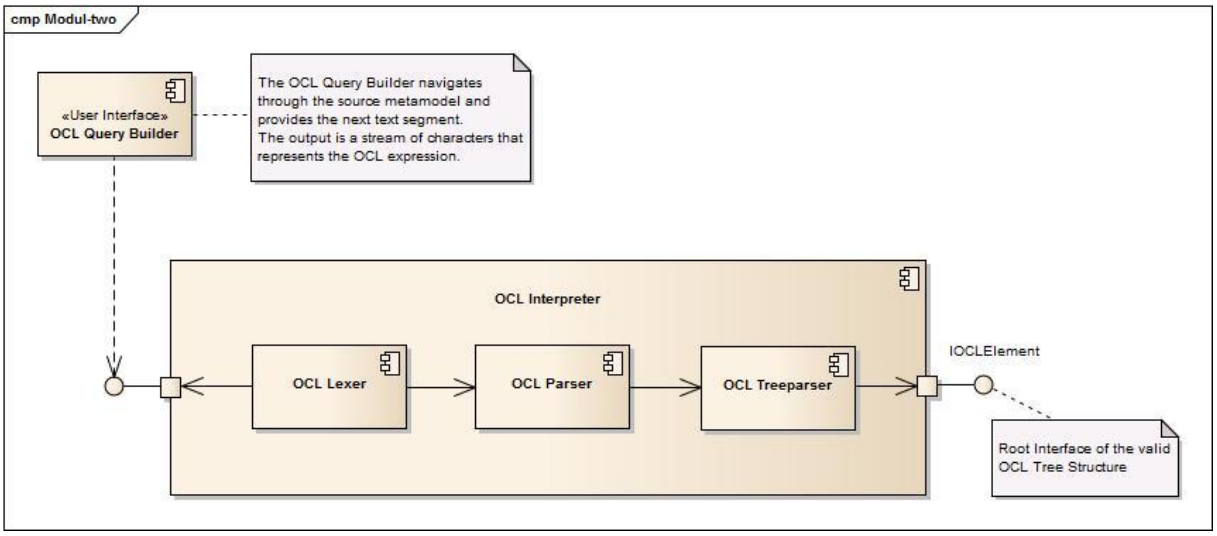

Figure 22 – OCL interpreter

The parser generation was realized and tested with Eclipse Helios SR2[15] and the ANTLR plug-in (Version 3.2) [**44**]. The tools can be downloaded and are explained by several tutorials and showcases in a detailed manner.

4.2.2.1 Modified OCL grammar

It is a well-known fact, that the OCL grammar as defined in the language specification is ambiguous and is not suitable for a parser generator [**45**]. The specification uses production rules, which are not available during a purely syntax based analysis (such as parsing). As the authors of the paper [**45**] mentioned, "The disambiguating rules depend on information from the environment, i.e. semantic information from the user model and context of the expression." The generated AST is validated against the environment information during the transformation process described in the next section 4.2.3 to minimize the ambiguities. The defined grammar of the ADOCLE is an equivalent grammar to that defined in the OCL specification.

[15] Eclipse Helios is the annual release of Eclipse projects that provide a development environment. http://www.eclipse.org/helios/

4.2.3 OCL Transformator

The third module represents the core of ADOCLE. The task of the core is to generate a semantically equivalent expression in a target metamodel for an OCL expression. The significant point of the task is the variable target metamodel and the implementation of the target metamodel. A metamodel could be designed as a tree in XML like the UML metamodel (see [**1**], [**2**]), as schema in a relational database (see [**5**]) or as an object structure in any programming language. ADOCLE has to provide an abstract strategy to support any kind of metamodel. First, a Strategy Pattern [**46**] is applied to manage different transformation algorithms depending on the target metamodel and the realization of the target metamodel. A target metamodel could be implemented in different ways. In the case of the ADOCLE prototype, an OCL2SQL transformation has been determined according to the physical database schema of the EA, which supports quite a lot of databases and several versions of these databases. But, the interpretation of SQL depends on the driver, which provides the access to the database management system. Therefore the OCL2SQL transformation is based on the SQL-92 standard [**7**] and database specific functions are replaced by a dialect concept, which helps generating optimized queries to those specific versions of database. Figure 23 shows the approach of the ADOCLE that implements a generic transformation between OCL and SQL for the automatic execution of OCL rules directly in the EA database. Transformation strategies for other CASE tools can be added as autonomous algorithm.

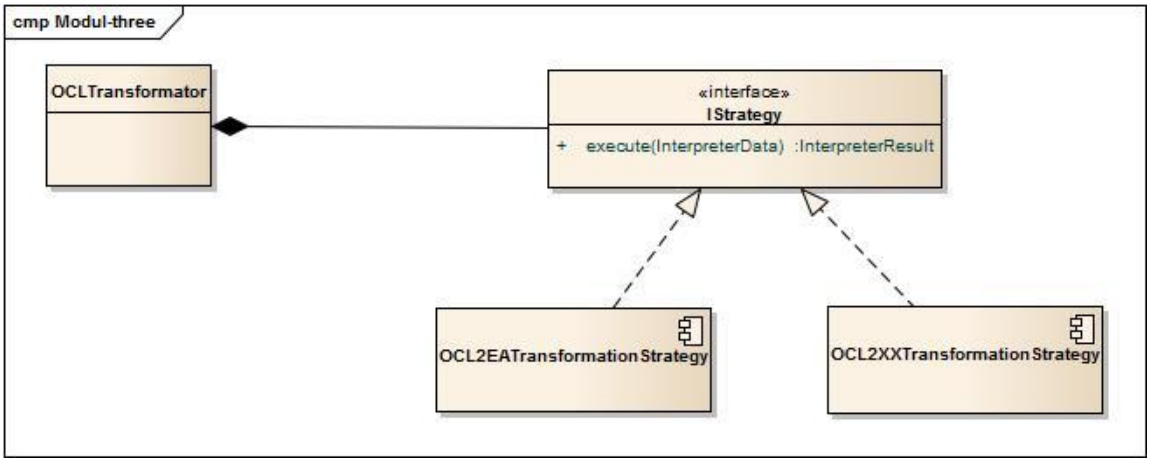

Figure 23 - Module three - Strategy Pattern

The approach seems simple. But how can every single part of the object tree generated by the OCL treeparser (see Figure 22) generate an equivalent part of the target metamodel. Figure 24 shows a combination of an interpreter pattern and a command pattern to act after receiving a part of the AST to transform it in a target metamodel element [**46**].

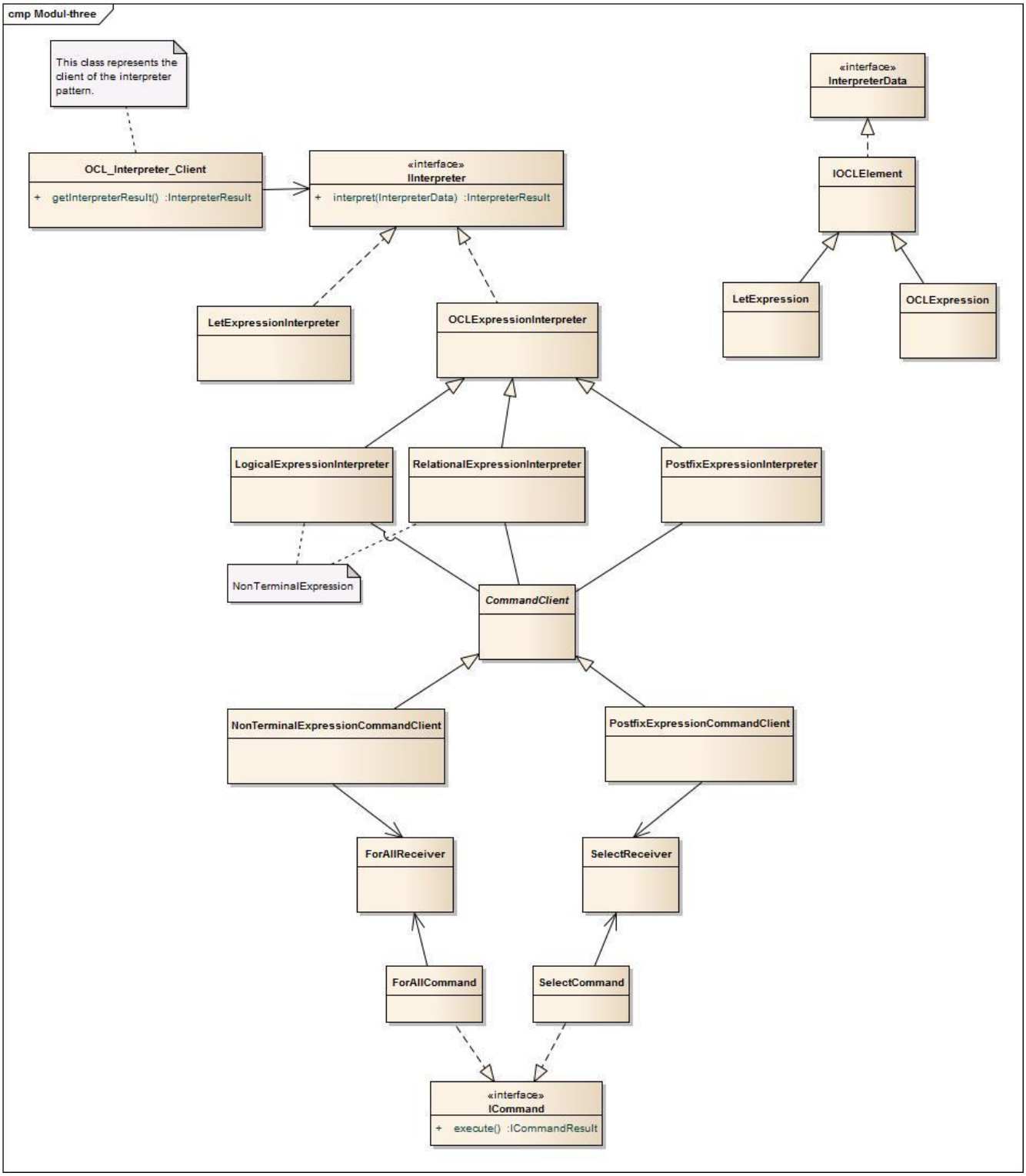

Figure 24 - Design Patterns used for the OCL Transformator

The interpreter pattern takes the object tree generated by the OCL treeparser and produces an independent result object, which represents the provided OCL expression. Depending on every object in the OCL tree, the suitable interpreter is used to generate a result object. The final result object contains the equivalent expression of the target metamodel and the result of the executed expression. A detailed illustration of the behaviour of the OCL transformation strategy is described in section 4.2.3.1.

The command pattern is activated within the suitable interpreter in the following cases:

(1) The suitable interpreter has to execute an operation of the OCL specification [**3**]. The OCL operations listed in the OCL Specification [**3**] cannot be identified as separate element by the OCL parser and the OCL treeparser, because the operation depends the type (see Chapter 2.2.4). The module two only guarantees that the input is valid. If an operation is recognized, a command is executed to manage the operation depending on the type. The identification of an operation is based on the OCL tree. The previous element has to be a right arrow ("->") (only special cases allow a dot (".")) and the following element has to be round brackets with or without parameters. The list of supported operation was limited to those operations that are necessary for the test cases or very commonly used operations (see Table 9).

(2) The result of an expression of any kind has to be resolved. For instance a relational expression has a left and a right term and an operator. The result type of a relational expression is Boolean. The execution of the relational expression to generate the Boolean value is managed by a command. The command is executed after resolving the left and the right term of the expression. The terms can also be large OCL sub trees corresponding to the OCL grammar.

4.2.3.1 Transformation example of a simple OCL constraint

The transformation example explains the steps how the suitable interpreter is chosen and a command is executed. Therefore the OCL rule in Figure 25 is taken, which is part of the UML Specification [**2**] and is also part of the evaluation (see Table 15). The rule expresses that a final state of a state machine diagram is not allowed to have any outgoing connectors. Otherwise the state machine model is invalid.

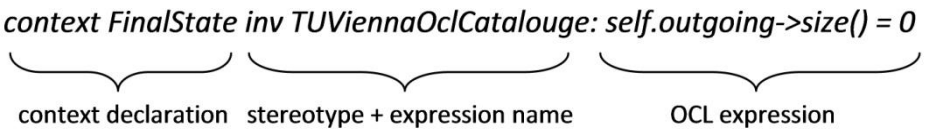

Figure 25 - OCL constraint basic structure

As input for the transformation process the OCL parser delivers an OCL tree. In Figure 26 the output for the above sample is depicted. Figure 26 is generated by the ANTLR plug-in of Eclipse Helios SR2.

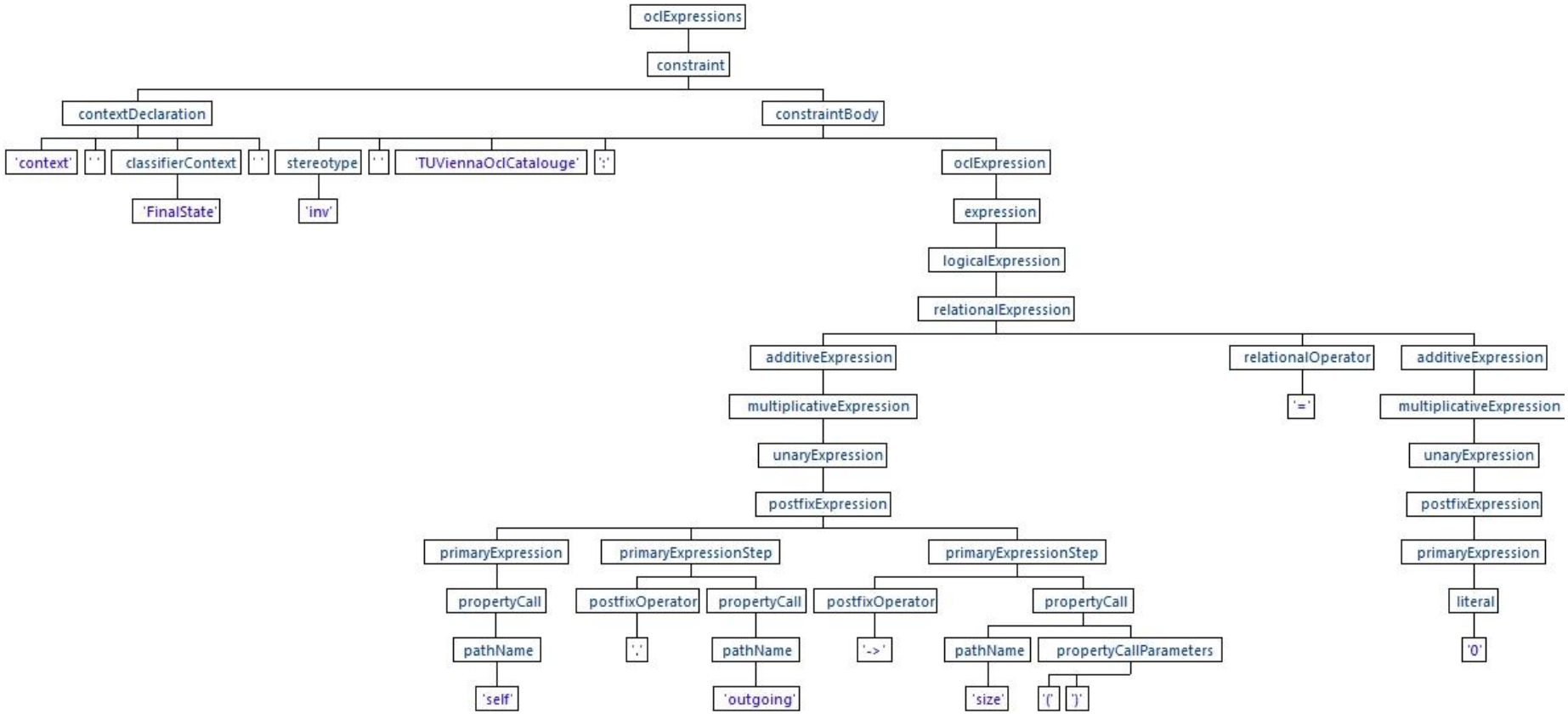

Figure 26 - OCL Parser Output for the example

During the transformation process (see sequence diagram - Figure 27), the first step is to set the context declaration. The global variable *self* is set to the *classifierContext* (see Figure 26) – which is an object of the source metamodel. In the illustration *self* is set to *FinalState*. The stereotype and the expression name has no deeper meaning for the instance, but the OCL parser need the parts to validate the input as correct.

The OCL expression in Figure 25 represents a relational expression – OCL tree. Due to the type of the expression – usage of the interpreter pattern - the corresponding interpreter is identified to resolve the expression type. The interpreter separates the relational expression in two postfix expressions (left and right term) and the relational operator equal. These types are also resolved by suitable interpreters. This behaviour is applied until the leaves of the OCL tree are reached. Each interpreter generates a result object for its level. In the end the result of the relational expression is provided. This behaviour works for the most objects of the OCL tree. Expectations are managed by commands.

The relational interpreter starts the interpretation of the left and right term with two instances of a *PostfixExpressionInterpreter* (object postfix1 and postfix2 - see Figure 27). The left term is the more interesting one, because it navigates through the source metamodel and then executes an operation. Therefore every step of the postfix expression is resolved using different resolving approaches.

The first element builds the base for the equivalent expression in the target metamodel. Typically it can be described as a set of elements. For instance, all final states of an modelled state machine. All further steps excluding the last one provide interim set results. Such steps can be navigations through the metamodels or operations that constraint or expand the previous set. For this example, there is only one navigation step. So in Figure 27 the loop only contains the resolver for navigations. The object single element resolver and navigation resolver (see Figure 27) are responsible for the SQL generation based on the mapping rules. Details about the algorithm will be explained in Chapter 4.2.3.3 and 4.2.3.4.

The last step packs generally the result object for the postfix expression. In this OCL expression example the operation *size* is identified within the last step, which activates the command client to create a SizeCommand. The function of the OCL command *size* is generated and executed in the target metamodel. The result object is delivered up to the relational expression, which represents the resolution of the left term in the target metamodel.

The right term is quite simple. The literal is identified and transformed in an interim result object.

If the left and the right term are resolved, the result of the relational expression has to be created. Therefore a command is used to compare the result objects of the right and the left term using the given operator. In the example the result is of type Boolean, which describes if the state machine is valid or not.

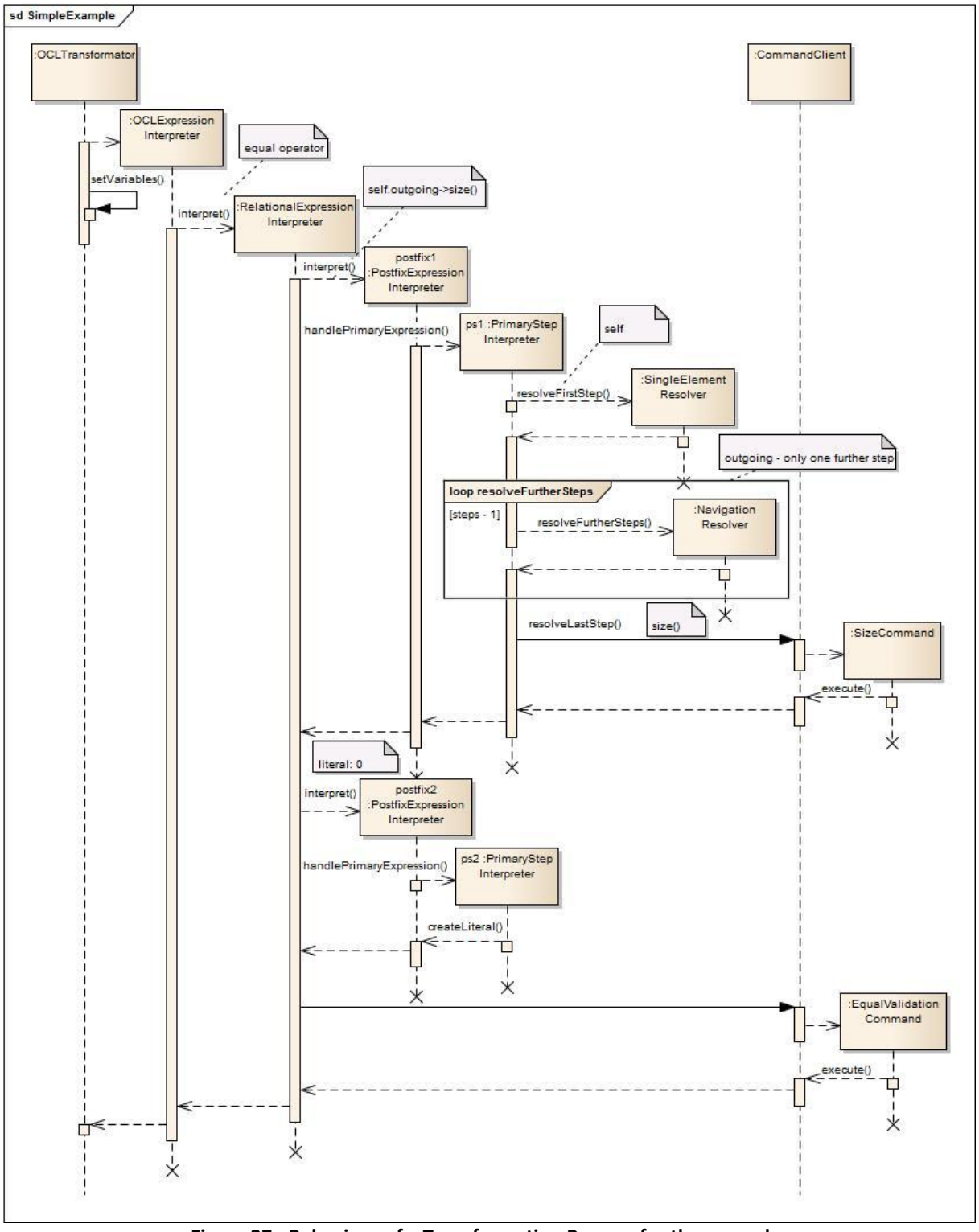

Figure 27 - Behaviour of a Transformation Process for the example

4.2.3.2 Transformation of complex OCL constraint

More complex OCL rules may contain constraints with sub expression (see Figure 28). A sub expression is a sub tree of the whole OCL tree created by the OCL parser.

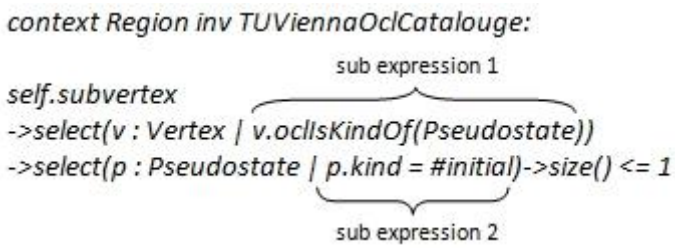

Figure 28 - Complex OCL example

Such a sub tree leads to a separate execution of the transformation process. The approach generates an equivalent expression of the sub expression as an executable instance, which represents a set of elements. A big advantage is that result set operations can be applied, which makes the implementation a lot easier to understand. The Chapter 4.2.3.3 and 4.2.3.4 show different algorithms for the OCL/SQL transformation and contains a description why a separate execution achieves more benefits.

Moreover, sub expressions within an operation can be transformed in a concurrent way. The concurrent programming approach is not yet used in the prototype, because the test cases of the state machines does not often use sub expressions (see Chapter 5.4.2).

4.2.3.3 Constraint Algorithm

The principle of the constraint algorithm is to take a basic set of elements, for instances all states in a state machine diagram and constraint or expand the set of elements. The result set is described with a select statement in SQL [**7**] (see Chapter 3.1.12.2.2). The basic set of elements is created by the single element resolver. It takes a source metamodel element (see Chapter 4.2.3.1), resolve the mapping rules and generate a select statement that includes required conditions to represent a result set of the chosen source metamodel element.

When the OCL expression is processed, the source metamodel is navigated, starting at the context element defined in the OCL expression. Based on the mapping model between the source metamodel (UML metamodel) and the target metamodel (database schema of EA), the necessary DB tables are collected for the FROM clause. The conditions in the WHERE clause are derived from the mapping patterns and added to the same select statement. Joins between tables, derived from the corresponding association mapping pattern (see Chapter 4.2.1.2.4), are also added as condition in the WHERE clause. Special kinds of joins like UNION or INTERSECT are not possible with the constraint algorithm, because we

just constraint or expand the set of elements using the WHERE clause and does not compare two different result sets.

The qualifiers (table identifier, variable name, etc.) are created from a central generator that controls the names of all instances and the related target metamodel elements, in this case the tables of the database schema to guarantee the singularity of the qualifier.

The constraint algorithm led very quickly to valid results for simple cases. The advantage is that a condition for the set can be easily added. But the complexity of OCL expressions is much higher than anticipated. During the implementation of the constraint algorithm following issues were noted:

- It is possible to add conditions that are mutually exclusive.
- The power of different kind of join cannot be used efficient.
- Set operations are not possible in every case.
- Set operations are hard to identify in the output select statement.

Some parts of the constraint algorithm could be reused for another solution approach, for instance the *Subselect* algorithm described in the next section.

4.2.3.4 Subselect Algorithm

The name of the algorithm already describes the concept of the *Subselect* algorithm. The single element resolver works as in the constraint algorithm; it generates a select statement that represents a set of elements resolving the metamodel mapping. This select statement could be used as sub select in surrounding select statements for set operations, complex SQL constructs or any kind of join could be specified.

Figure 29 and Figure 30 leads through the *Subselect* algorithm steps for navigations through the source metamodel and give an introduction to the algorithm. The figures show how the OCL expression starting at *Pseudostate*, navigating to *Transition* and end at *Trigger,* is resolved.

For navigating through the source metamodel the algorithm generate a select statement for each source metamodel element. It starts at the initial point and create the first select statement (see Figure 29 – set A). The next select statement is produced for the end of the first navigation step (see Figure 29 – set B). These two select statements are set as sub select statements in a surrounding select statement (see Figure 30). The used association between *Pseudostate* and *Transition* is called *outgoing*, which is described with some kind of join that is added to the surrounding select statement (see Figure 29 – set C). The output of the navigation resolution is again a select statement that can be used for further steps.

The next navigation from *Transition* to *Trigger* facilitates the association *trigger*. The *Subselect* algorithm generate a select statement that represents the set of triggers (set D) and join the previous select statement (set C) with the set of triggers (set D) in a surrounding select statement (set E).

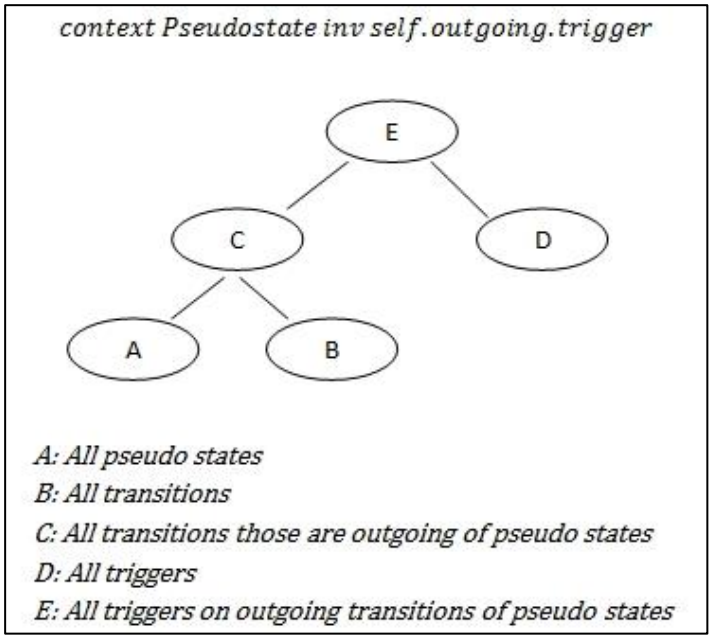

Figure 29 - Subselect algorithm example

The final output of navigations in the source metamodel is some kind of SQL tree. The leaves represent basic sets, which is source metamodel subset, intermediate sets displays the join between sets and the root outlines the final set.

The approach is a bottom-up construction that can reuse the generated sets within a SQL tree. In contrast of the constraint algorithm set operations or more complex operations can be defined due to the surrounding select statement. The set operations and other SQL constructions are needed to express parts of the OCL grammar [**3**]. The Chapter 4.2.3.5 lists the range of already implemented features and Chapter 4.2.3.6 explains transformations where set operations or complex constructions are needed.

The resulting construction of any SQL tree is again a select statement that represents the current set surrounding previous navigation steps, set operations or other complex constructions. A big advantage of the *Subselect* algorithm is that the small sets within the tree illustrate the elements very well and can be executed faster than huge SQL statements with more joins.

Figure 30 shows a generated output of the above mentioned OCL expression. Typically the select statement only includes necessary columns like identifiers or constraining columns. The identifiers are needed to join the generated select statements in the surrounding select statement. The qualifier naming generator use a pattern for name creation; it takes the name of the source and the target metamodel that are mapped, adding a token at the beginning. The names of the sub select statement are sorted alphabetically.

```
//set E
SELECT DISTINCT d.Trigger_guid FROM
        //set C
        (SELECT DISTINCT b.ea_guid, b.Start_Object_ID FROM

                //set A
                (SELECT DISTINCT S1_pseudostate_t_object.Object_ID, S1_pseudostate_t_object.ea_guid,
                S1_pseudostate_t_object.ParentID FROM t_object as S1_pseudostate_t_object
                WHERE ((S1_pseudostate_t_object.NType = 0 OR S1_pseudostate_t_object.NType = 1 OR
                S1_pseudostate_t_object.NType = 10 OR S1_pseudostate_t_object.NType = 11 OR
                S1_pseudostate_t_object.NType = 12 OR S1_pseudostate_t_object.NType = 13 OR
                S1_pseudostate_t_object.NType = 14 OR S1_pseudostate_t_object.NType = 15 OR
                S1_pseudostate_t_object.NType = 3 OR S1_pseudostate_t_object.NType = 5) AND
                (S1_pseudostate_t_object.Object_Type = 'EntryPoint' OR
                S1_pseudostate_t_object.Object_Type = 'ExitPoint' OR S1_pseudostate_t_object.Object_Type
                = 'StateNode' OR S1_pseudostate_t_object.Object_Type = 'Synchronization'))) as a,

                //set B
                (SELECT DISTINCT S1_transition_t_connector.ea_guid,
                S1_transition_t_connector.Start_Object_ID FROM t_connector as S1_transition_t_connector
                WHERE S1_transition_t_connector.Connector_Type = 'StateFlow') as b

                //join between set A and B
                WHERE (a.Object_ID = b.Start_Object_ID)) as c,

        //set D
        (SELECT DISTINCT LEFT(M1_trigger_t_xref.Description, 255) AS Trigger_guid, M1_trigger_t_xref.Client
        FROM t_object as S1_trigger_t_object, t_xref as M1_trigger_t_xref WHERE
        S1_trigger_t_object.Object_Type = 'Trigger' AND M1_trigger_t_xref.Behavior = 'trigger') as d

        //join between set A and B
        WHERE (c.ea_guid = d.Client)
```

Figure 30 - Subselect algorithm SQL

4.2.3.5 Range of functionality

In the case of the development of a prototype not all parts of the OCL grammar [**3**] are supported already. The EBNF is fully provided from the OCL parser and tree parser, but not all parts are triggered in the transformation process to generate SQL. The supported functions (represented by the following tables) crystallize out of the required functionality managing the test scenarios and the work of Alexander Schmidt [**29**] (see Chapter 4.2.3.6).

Expression	Description	Example
OCL constraints	Describes a list of constraints	
Constraint	Defines the context declaration and the constraint bodies	
Context declaration	Describe the context declaration – only the class context is supported, the operation context is not trigger for the transformation process.	
Classifier context	Part of the Context declaration which sets the class context explicit.	Context State
Constraint Body	Defines the stereotype of the constraint and includes an OCL expression.	
Stereotype	The stereotype describes when the OCL expression has to be valid.	inv, pre, post
OCL expression	The OCL expression can include a list of let expressions (not supported) and one concrete expression.	
Expression	Super type of all kind of following expressions	
Logical expression	A logical expression consists of a left and a right argument and a logical operator.	self.kind = #entryPoint or self.kind = #exitPoint
Relational expression	A relational expression consists of a left and a right argument and a relational operator.	self.size() = 1
Unary expression	A unary expression consists of an unary operator followed by an expression.	not (self.isEmpty())
Postfix expression (Navigation)	Navigation through attributes, association ends, association classes, and qualified associations.	self.connectionPoint
Postfix expression (Operation)	Use of an operation (see Table 9)	self->size()
Literal	Literals characterize the different kind of possible basic types.	Boolean, Integer, Real, String
Enumeration Literal	Each kind of the enumeration is identified as an enumeration literal.	self.kind = #entryPoint
Return Type	Some operations delivers a result sets. The return type can be defined.	
Declaration	Within operations variables can be declared with a type to express easier rules.	self->select(s1, s2 : State | s1 <> s2)

Table 7 - List of supported OCL expressions

Only few parts are not supported:

- Operation context
- Let expressions
- Additive or multiplicative expressions
- If-Then-Else expressions
- Identification of Bag, Collection, Sequence or Set

Operator	Description	Type
and	Boolean algebra: True if the left and the right argument are true Set operation: Intersect of the left and the right set	Logical operation
or	Boolean algebra: True if the left or the right argument is true Set operation: Union of the left and the right set	Logical operation
xor	Boolean algebra: True if the left and the right argument are the same Set operation: Combination of operators: and, or and not	Logical operation
implies	Boolean algebra: True if the left and the right argument are Set operation: Combination of joins and aggregate functions (see XX)	Logical operation
not	Boolean algebra: True if the expression is false and otherwise Set operation: Negation of the set	Logical operation
=	True if the left and the right argument are the same	Relational operation
<>	True if the left and the right argument are not the same	Relational operation
<	True if left argument is lower than the right argument	Relational operation
<=	True if left argument is lower than the right argument or the arguments are the same	Relational operation
>	True if left argument is greater than the right argument	Relational operation
>=	True if left argument is greater than the right argument or the arguments are the same	Relational operation

Table 8 - List of supported OCL operators

Operation	Description	Type
oclAsType(type)	The current element as of the given type	AfterDot-Operation
oclIsTypeOf(type)	True if the current element is an instance of the given type	AfterDot-Operation
oclIsKindOf(type)	True if the current element conforms to the given type	AfterDot-Operation
size()	Number of elements in the collection	Standard-Operation
isEmpty()	Does the collection contain no element?	Standard-Operation
notEmpty()	Does the collection contain one or more elements?	Standard-Operation
exists(expression)	Has at least one element for which expression is true?	Iteration-Operation
forAll(expression)	Is expression true for all elements?	Iteration-Operation
reject(expression)	Returns a collection containing all elements for which expression is false	Iteration-Operation
select(expression)	Returns a collection containing all elements for which expression is true	Iteration-Operation

Table 9 - List of supported OCL operations

4.2.3.6 Transformation of OCL operators and OCL operations

In a more basically work [**29**], the author Alexander Schmidt described transformation patterns between OCL and SQL formal and classified them in more or less critical translation rules. The classification in [**29**] led to a restriction of the supported features of the prototype for the first development cycle. As Alexander Schmidt mentioned there are mapping problems for OCL operations like the feature iterate. On the other hand some of the mapping problems are not commonly used operations and a workaround using other OCL expressions instead is possible. It is not necessary to implement all OCL operations to provide the whole functionality.

In this work the author figured out that the transformation rules are dependent on the database specification and their functionality. For example INTERSECT of SQL-92 standard is not implemented as keyword in all commercial databases like Microsoft Jet Database Engine[16]. As mentioned above a dialect concept has to be included to exchange keywords or workaround methods.

The work [**29**] was essential for this master thesis, because the transformations patterns constitute a scientific working base and the transformation rules are also referenced in other scientific works or are used in projects. The framework described in [**47**] follows quite similar approach to the idea of the ADOCLE. It is generating query language code for OCL invariants. The concept is module-based and uses an abstract syntax tree for describing the UML standard, but differs in the OCL/SQL translation method and the layer of metamodelling [**1**] to ADOCLE. In the approach of [**47**] models on layer M1 are translated in database-specific DDL (Data Definition Language). The OCL invariants are mapped to SQL Query views

[16] The **Microsoft Jet (*Joint Engine Technology*) Database Engine** is a database engine on which several products of Microsoft (Access, Visual Basic, etc.) have been built.

and the validation is executed manually. This framework is a fundamental part of the Eclipse-Plugin DresdenOCL (see Chapter 6.2.1).

For ADOCLE, the transformation rules in [**29**] has been implemented and tested using the described combination of interpreter and command pattern in Chapter 4.2.3.1. In most cases the patterns delivered the desired results like the logical and relational operators excepting *implies* (see Chapter 4.2.3.6.1). The supported operations (see Table 9) could be transformed with some transformation patterns from [**29**] like *forAll* or *exists*, others are implemented in own constructions like the *isEmpty*, *size* or *select*. The implementation of *isEmpty* or *size* was much easier using a source code validation as a transformation in SQL followed by an execution. Both returns equivalent results.

Nearly all transformation patterns were a huge help for the implementation of ADOCLE, but the *implies* operator has to be implemented in its own way.

4.2.3.6.1 Transformation of the implies operator

When we talk about the concept of implication, it is necessary to distinguish between different meanings. In the propositional logic an *implies* connective expresses a binary function, representing the following truth table [**48**], [**49**]:

A	B	A ⇒ B
T	T	T
T	F	F
F	T	T
F	F	T

Table 10 - Implication truth table

$A \Rightarrow B$ is an abbreviation for $\neg A \vee B$ in classical logic. It cannot be extended to more than two arguments. It has the meaning "if A is true, then B is also true" [**48**], [**49**]. The transformation pattern in [**29**] match with this definition, but the resolution of $\neg A \vee B$ does not lead to the required results.

The *implies* operator can be considered as a symbol of the formal theory [**48**]. The symbol used to denote *implies* is $A \models B$, $A \supset B$ (A is a real superset of B). It is also called entailment, logical implication, semantic implication, logical consequence, etc. Because it relates to the model theory saying that every model/interpretation of A is also a model of B, this definition is stronger than the binary function in the propositional logic. A logical consequence like $A1, A2, \ldots, Ak \models B$ states that whenever $A1, A2, \ldots, Ak$ is true, then B must be true as well. The conclusion follows deductively from the premises. This does not mean that the conclusion is true, as the premises can be false [**50**].

Based on an analysis of the definitions of *implies* the OCL operator *implies* has a structure like $A1, A2, \ldots, Ak \models B1, B2, \ldots, Bk$. A typical OCL expression of the UML Specifi-

cation [**1**], [**2**] define rules like $A1, A2, \ldots, Ak \mid = B$ or $A \mid = B1, B2, \ldots, Bk$. The interpretation of the logical consequence leads to the following approach. As first step the premises $A1, A2, \ldots, Ak$ are formed to a basic set, which is used for the validation of each expression $B1, B2, \ldots, Bk$ in the conclusion. The condition of an expression in the conclusion is transformed to an aggregate function within a select statement. Therefore a combination of the SQL elements: LEFT JOIN, GROUPBY and a HAVING clause helps to validate every data record in the basic set. If the conclusion has more expressions every generated select statement is validated according to correctness and is then combined dependent on the operator set between them. A small example out of the test cases (see Chapter 5.4.2.5) demonstrates the construction and the record sets of the database that contains two expressions in the conclusion. The OCL expression describes that in a complete state machine, a join vertex must have at least two incoming transitions and exactly one outgoing transition. In OCL it is expressed as Figure 31 shows.

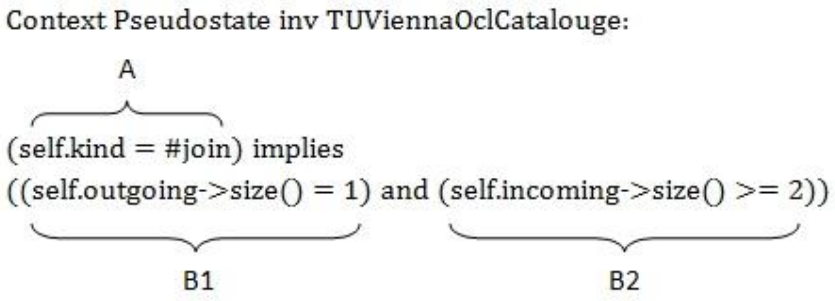

Figure 31 - Implies example OCL expression

In the defined test data only one join element exists. Table 11 shows the basic set record sets that influence the validation expressions B1 and B2. The record sets of the validation expressions B1 and B2 are listed in the Table 12 and Table 13.

Object_ID	ea_guid	ParentID
303	{411F8899-8284-4380-B4FB-34D477ECC322}	0

Table 11 - Implies example set A elements

ea_guid	amountOfOutgoingTransitions
{411F8899-8284-4380-B4FB-34D477ECC322}	1

Table 12 - Implies example set B1 validation

ea_guid	amoutOfIncomingTransitions
{411F8899-8284-4380-B4FB-34D477ECC322}	3

Table 13 - Implies example set B2 validation

The amount of outgoing transitions is equal one and the amount of incoming transitions is greater than two, so the tables contain the record set. If the amount of outgoing transitions

is not equal one, Table 12 will not contain the dataset. After the intermediate results are calculated, the amounts are checked depending on the logical operator between. Due to the *and* operator the record sets of the validation expressions have to be the amount of the original set A, otherwise the validation is false. The validation expressions B1 and B2 are interpreted as true and the select statements of B1 and B2 are joined dependent on the logical operator between them. The *and* operator leads to a join of the select statements of B1 and B2 using an intersect statement. For instance, the *or* operator would lead to join using a union statement. The amount of the result does not change because both contain the same element and the conclusion is interpreted as true. The OCL expression of Figure 31 is valid because the validation expressions B1 and B2 are interpreted as true and the joined select statement is interpreted as true.

The solution approach uses an abstract syntax tree for the interpretation of the conclusion. Each expression $\mathrm{B1}, \mathrm{B2}, \ldots, \mathrm{Bk}$ is a node within this tree. Depending on the logical operator between the nodes the interpretation of the tree leads to a Boolean result and the suitable select statement. All test cases delivers the required results if an *implies* connective is included in the OCL expression.

The third module provides the ability to transform and interpret a single OCL expression. This feature can be used for an interactive OCL console like Eclipse MDT provides. The fourth module complements the ability for more than one OCL expression.

4.2.4 OCL Validator

The fourth module is responsible for the automatic validation of models. Each drawn model within a supported modelling tool should execute all supported OCL expression of the UML Specification [**1**], [**2**]. The OCL validator takes the OCL expressions and lists all invalid elements with an error description for each invalid element that does not correspond to the UML Specification [**1**], [**2**]. The validator seems to be just an extension for the third module, but it is independent of the source or target metamodel. So it can take the list of OCL expressions and use any strategy that the third module provides. The result is always a list of invalid elements or a message for a successful validation.

4.3 User interface

This chapter illustrates the handling of the prototypical user interface. ADOCLE supports users by the creation of an OCL expression with a flexible OCL Query Builder using IntelliSense.

First, the models that should be validated have to be chosen. The user can define the location of its drawn models using the dialog in the toolbar. In the case of the ADOCLE development, the test data described in Chapter 5.4.1 are validated.

Next, the context has to be defined. The user navigates through the source metamodel to specify the entry point to a model element where the user wants to attach an OCL expression. As mentioned before in this chapter, the prototype is implemented for the UML metamodel (source metamodel) and the target metamodel represent the physical database schema of EA. The user goes step by step from the highest level package of the UML metamodel down to the model element, where the OCL expression should be defined. Therefore, the IntelliSense concept assists the user during the input. After pressing a dot, all sub elements are listed in a box, where the user can chose the desired one. In Figure 32 the context is set to *UML.StateMachines.BehaviorStateMachines.State* to show one of the test cases listed in Chapter 5.4.2. For the input of the OCL expression, the OCL Query Builder supports the generation of valid text segments using IntelliSense. The OCL Query Builder provides the navigation through the UML metamodel accessing attributes or association ends and possible operations according to the OCL type hierarchy. This feature is optional and can be disabled. In Figure 32, the following OCL expression is defined:

```
-- A simple state is a state without any regions.
context State inv TUViennaOclCatalouge:
(self.isComposite = true) implies
(not (self.isSubmachineState = true)
```

After pressing the *Execute* Button, the results of the created SQL expression are listed, as well as those elements that are not conform to a given OCL expression. The result sets are separated in two lists: valid results and invalid results.

When the OCL Query Builder IntelliSense function is chosen, the result lists are provided on the fly according to the current input of the OCL expression and the generated SQL expression out of the input. Thus, the user can additionally analyse the interim results of the current OCL expression input.

For development purposes, the generated SQL statement is shown as in the right upper part. In addition the source metamodel, the target metamodel and the manual mapping between are loaded and viewed in a tree structures. Finally, the mapping pattern based on the current context divided in the source metamodel elements, the target metamodel elements and the used elements of the metamodel concept are shown.

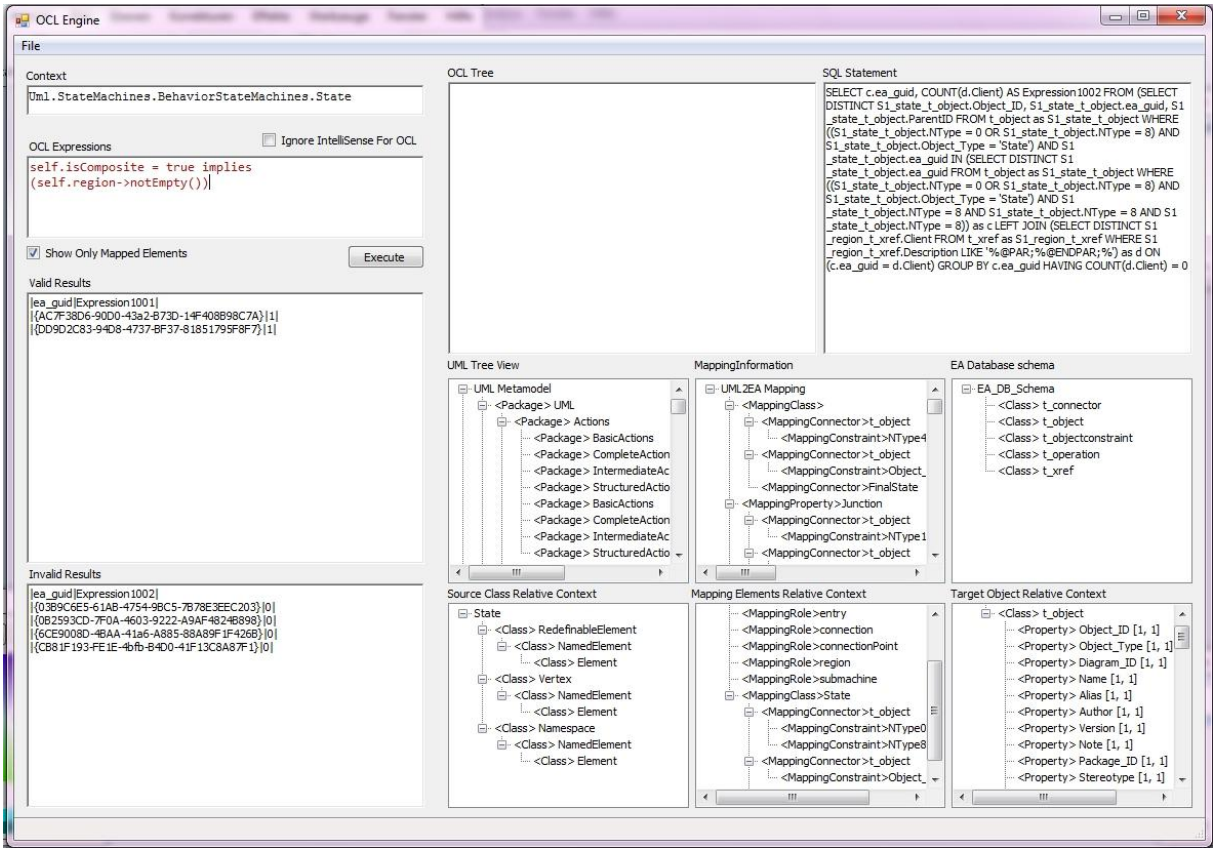

Figure 32 - ADOCLE UI

4.4 Development issues

For the sake of completeness, the development issues are documented. Enterprise Architect provides four wrapper APIs for its COM[17] interface (Java[18], Visual Basic (VB)[19], Delphi[20] and C#[21]). For an integration of ADOCLE as EA Add-In, the programming languages VB, Delphi and C# may be used. As development environment, Visual Studio 2010 extended with the add-ins NUnit[22] for testing and AnkhSVN[23] for the revision control is selected. The version details of the used tools follows:

- Visual Studio 2010 – Version 10.0.30319 RTMRel
- .NET Framework – Version 4.0.30319 RTMRel
- Enterprise Architect – Version 9.1.910, Database-Version 4.01

The source code is available on: https://subversion.assembla.com/svn/ocl-engine/

[17] http://www.microsoft.com/com/default.mspx

[18] http://www.java.com

[19] http://msdn.microsoft.com/en-us/vstudio/hh388573.aspx

[20] http://www.embarcadero.com/products/delphi

[21] http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx

[22] http://www.nunit.org/

[23] http://ankhsvn.open.collab.net/

5 Evaluation

The evaluation of the prototype shows which OCL expressions of the UML Specification [1, 2] are executable and acceptable quick for end-users. Therefore the OCL commands of the UML Specification are cut down to the state machine package of the UML metamodel, because the mapping between the UML metamodel and the Enterprise Architect database schema is also cut down to this subset (see Chapter 4.2.1.1).

5.1 Goal

The target of the evaluation is a performance analysis that compares the prototype using SQL commands and a solution approach of Sparx Systems.

As mentioned in Chapter 1.3, Enterprise Architect provides an API, which allows running programmatically through the model. This approach is manly used to provide validation rules within Enterprise Architect. Writing such rules requires the knowledge, how UML models are persisted in the EA database.

In the case of failing executions, the results are summarized and categorized in a consistency analysis for further implementation steps of ADOCLE.

5.2 Problem and hypothesis

For the experiment, the hypothesis is that the approaches present different amount of executable OCL expressions, amount of failing executions and different efficiency.

The advantage of the prototype is the performance, because the execution of SQL commands on a database schema is more efficient than an execution using a COM[24] hardware interface like the EA works with. The advantage of the approach of EA is that it is more powerful in describing OCL expressions and does not need complex mapping information. Of course, this is the main disadvantage of ADOCLE. Every element, association or other information of the source metamodel has to be mapped to the target metamodel. That means that the prototype is dependent on the target metamodel. If the target metamodel does not support the whole features of the source metamodel, it cannot be mapped and the OCL expression cannot be executed.

[24] http://www.microsoft.com/com/default.mspx

The measurement parameters:

1) amount of executable OCL expressions,
2) amount of failing executions depending on mapable OCL expression,
3) amount of failing executions depending on the expressive power of the target meta-model,
4) amount of failing executions depending on an OCL navigation to not focused subsets
5) and time in milliseconds

are computed by using unit tests. In failure cases, an exception is thrown to recognize the suitable failure. The measurement parameters are the base for the statistical hypothesis tests. The null hypothesis (H_0) is that ADOCLE delivers the results for the underlying OCL expressions faster than the solution of Sparx Systems. The alternative hypothesis says the opposite. Typically the test results are categorized in the following categories:

	Null hypothesis (H_0) is true	Null hypothesis(H_0) is false
Reject null hypothesis	False positive Type I error	True positive Correct outcome
Fail to reject null hypothesis	True negative Correct outcome	False negative Type II error

Table 14 - Statistical hypothesis categories

Based on this hypothesis, a consistency analysis (see Chapter 5.4) and a performance analysis (see Chapter 5.5) are conducted.

5.3 Test environment

The test cases (see Chapter 5.4.2) are executed on the following test environment.

MacBook Pro:

- Intel(R) Core(TM) i7-2635QM CPU @ 2.00GHz
- 4 GB main memory, Solid State Disk
- Windows 7 – Version 32 Bit Ultimate
- Visual Studio 2010 – Version 10.0.30319 RTMRel
- .NET Framework – Version 4.0.30319 RTMRel
- Enterprise Architect – Version 9.1.910, Database-Version 4.01

5.4 Consistency analysis

The consistency analysis concentrates on the evaluation of the correctness of the elements. This means, that the expected results are identical the effective results. If, there is a mismatch, it is documented, as well as all failing executions. Therefore, we used a traffic light concept based on the status of the test cases (see Chapter 5.4.2).

The status of a test case is marked green for a correct outcome (see Table 14). In the case of false positive or false negative results the status is marked red. The failing executions are divided in three subsets. The status of the test cases is marked yellow followed by one of these numerations.

1) The amount of failing executions depending on the expressive power of the target metamodel describe the expressive power of the metamodel mapping (see Chapter 4.2.1.1) and the consequences of a not mapable element. The OCL expression cannot be executed due to metamodel mapping problems.
2) The amount of failing executions depending on mapable OCL expression show the possibility of mapping OCL expressions to a target metamodel based on the implemented features of ADOCLE of the OCL Specification [**3**]. If a feature is used in the OCL expression that is not supported or not mapable for the target metamodel, it will be noticed by this category.
3) The amount of failing executions depending on OCL navigation to not focused subsets; in this evaluation the state machine subset of the UML Specification summarize the test cases that include other subsets. Probably the OCL expression leads to a successful result if the metamodel mapping will be extended in future steps.

For the consistency analysis, suitable test data is needed, which is illustrated in the following sections.

5.4.1 Test data

As first test data examples of the lecture “Model Engineering” are chosen (see Figure 33 and Figure 34). These examples describe small state machine that use a wide range of state machine components of the UML metamodel and are used as motivation examples to describe the components in detail.

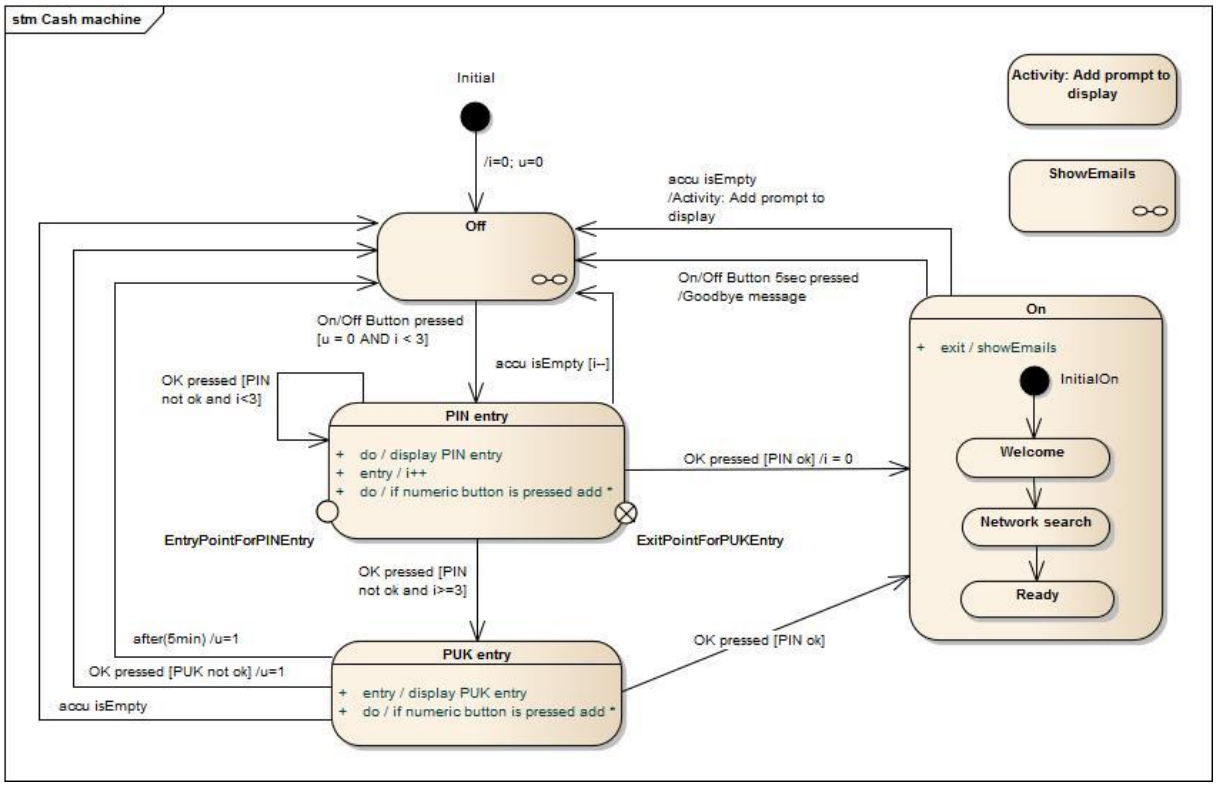

Figure 33 - Cash machine example - Test data 1

Figure 33 shows especially the different kind of transition constraints: behaviour, effect or guard. In Figure 34 the focus is set on states with regions and their transitions.

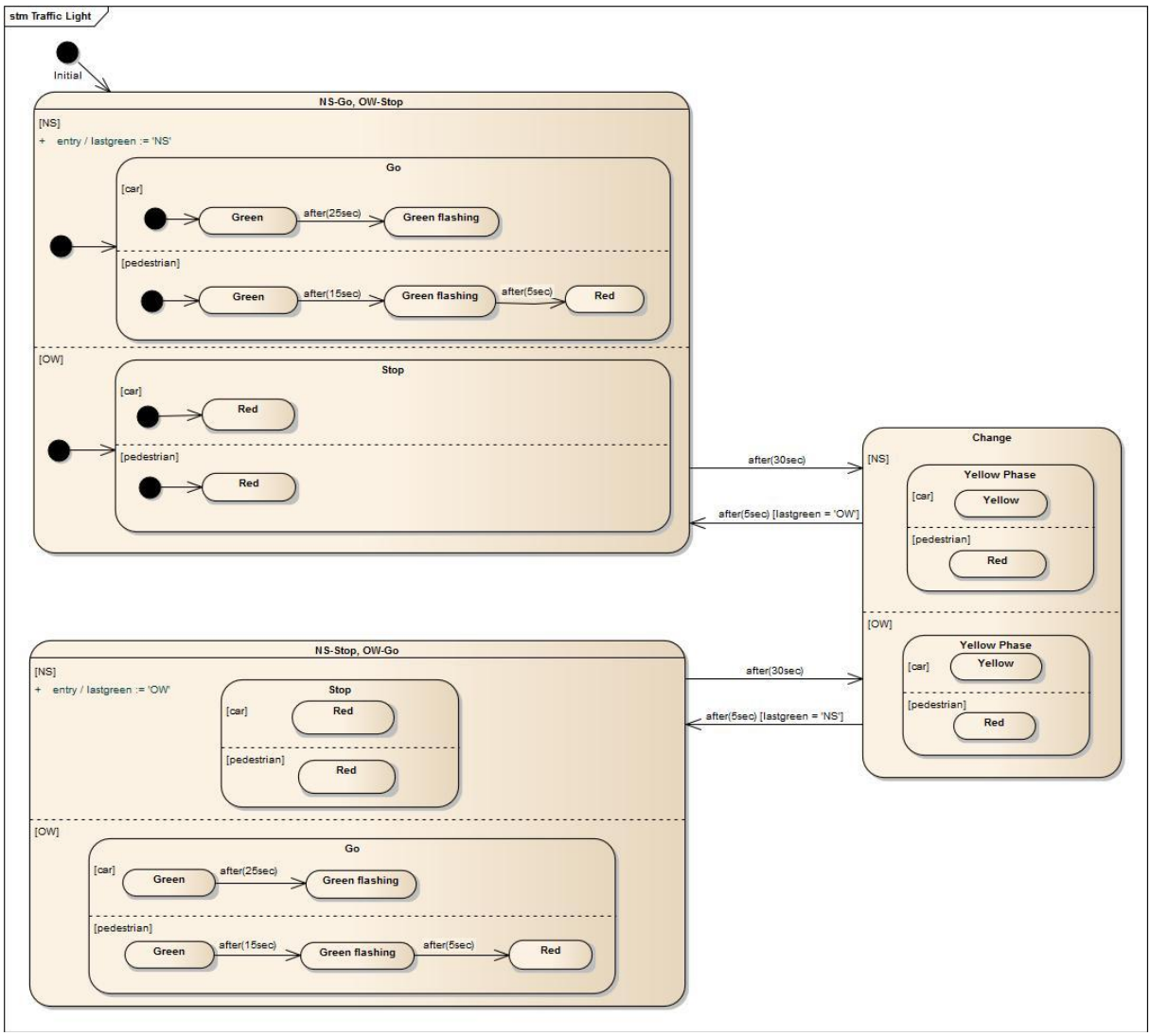

Figure 34 - Traffic light example - Test data 2

An example from the web enlarge the test data, to describe the connection point reference in a better way and describe the difference between a simple entry/exit point and connection point reference that is a entry/exit point.

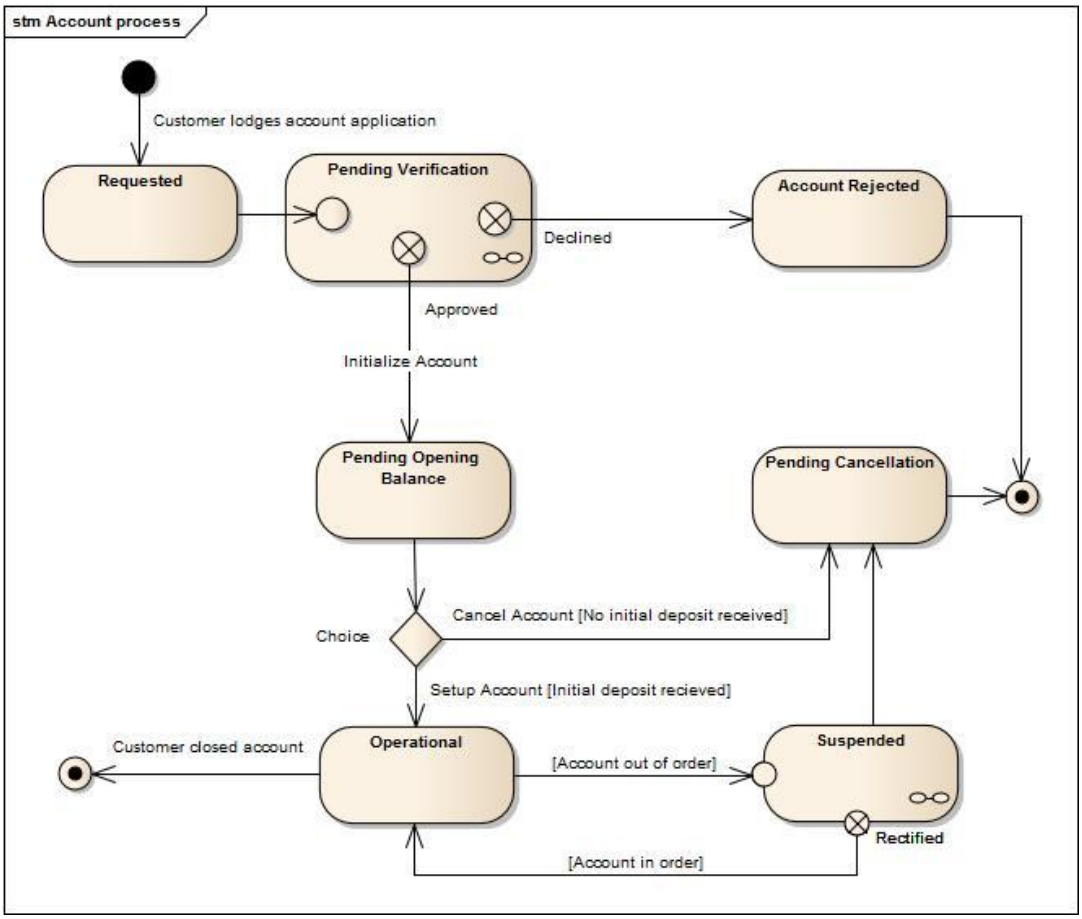

Figure 35 - Account process - Test data 3

The protocol state machines are already supported from the Enterprise Architect, but not all components of the UML specification. So a small example (see Figure 36) shows what is already provided.

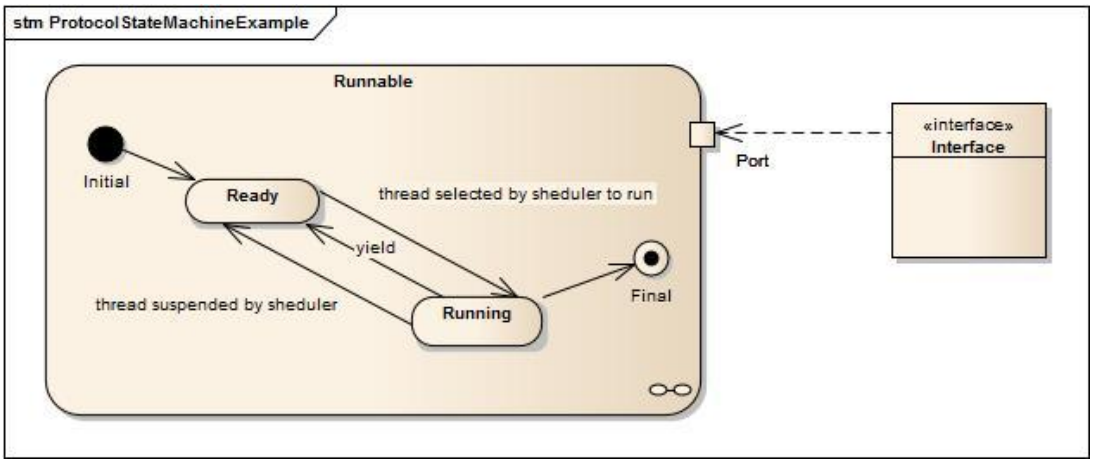

Figure 36 - Protocol state machine - Test data 4

The four examples above are drawn with the basic kind of state due to the basic configuration of the Enterprise Architect. No properties that the UML metamodel provides are used. It should demonstrate a model defined of a beginner and the validation errors against the well-formedness rules in the UML specification.

In the following example the different kind of transitions that are defined in the UML specification are explicitly set.

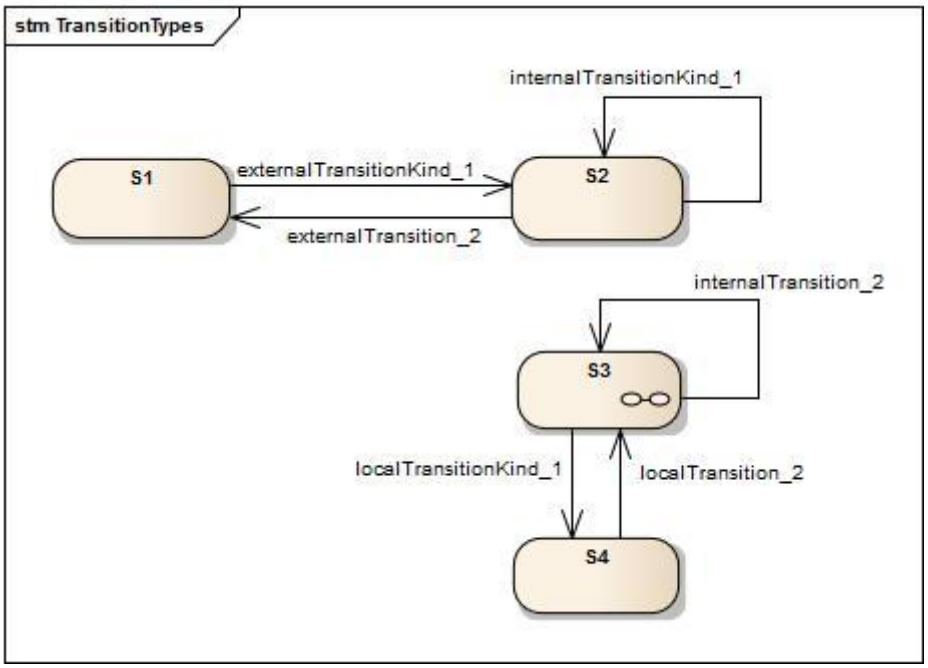

Figure 37 - Transition types - Test data 5

The next figure describes the different kind of states and completes the range of possible components of the state machine package.

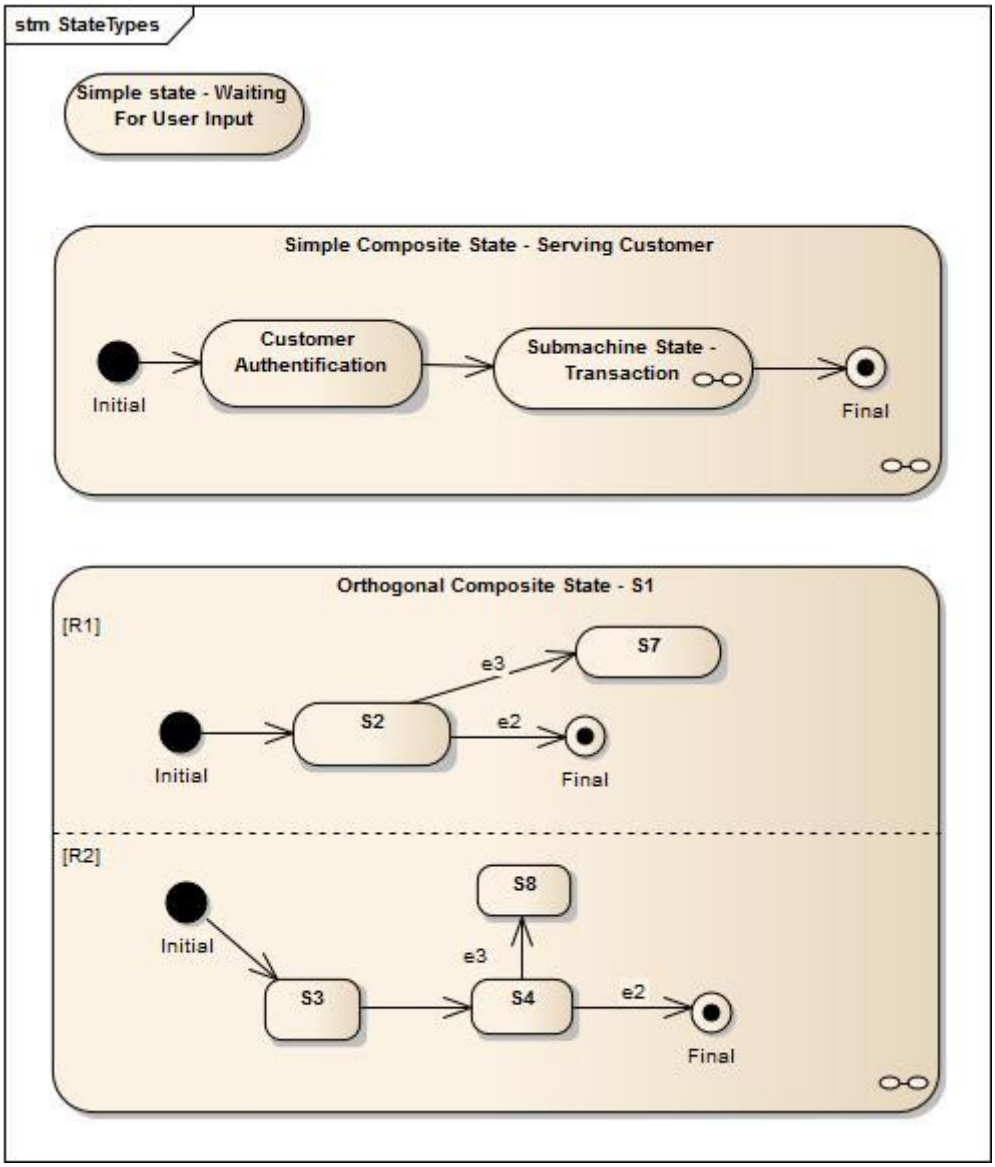

Figure 38 - State types - Test data 6

Finally a self-defined example displays all components of the state machine subset that are not listed in the previous examples.

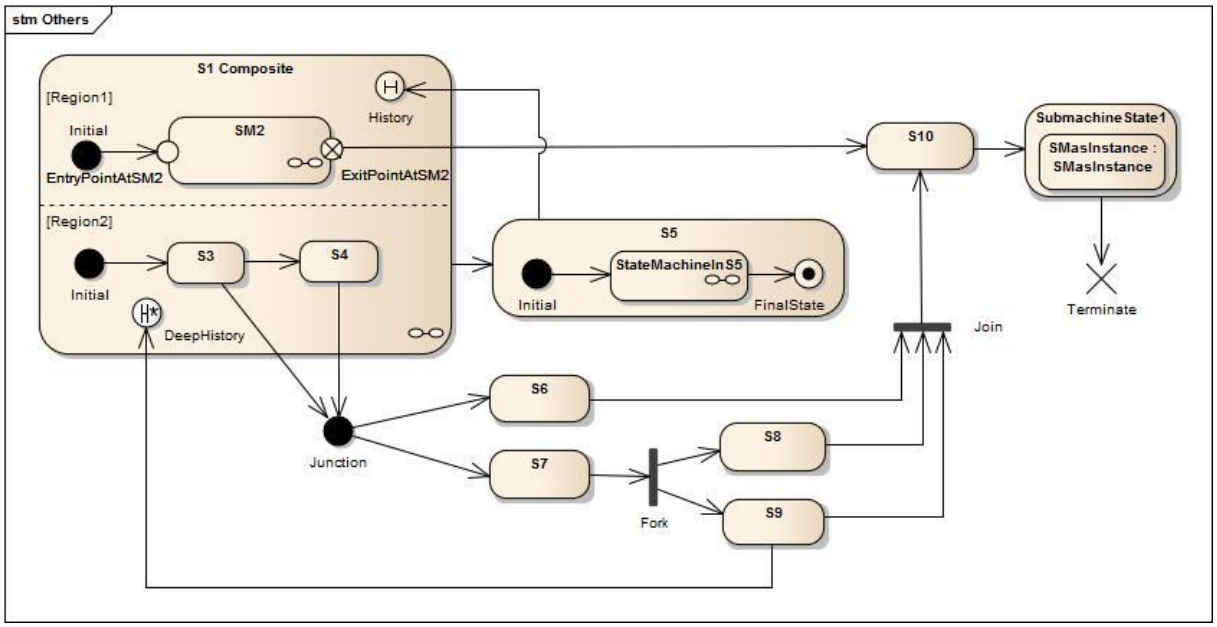

Figure 39 - Example for special state machine components – Test data 7

There is only one element that is already not part of the examples. The Time Event element is generally part of the activities package, but is also listed as part of the state machine package in the UML specification. For the sake of completeness an example is added to the test data.

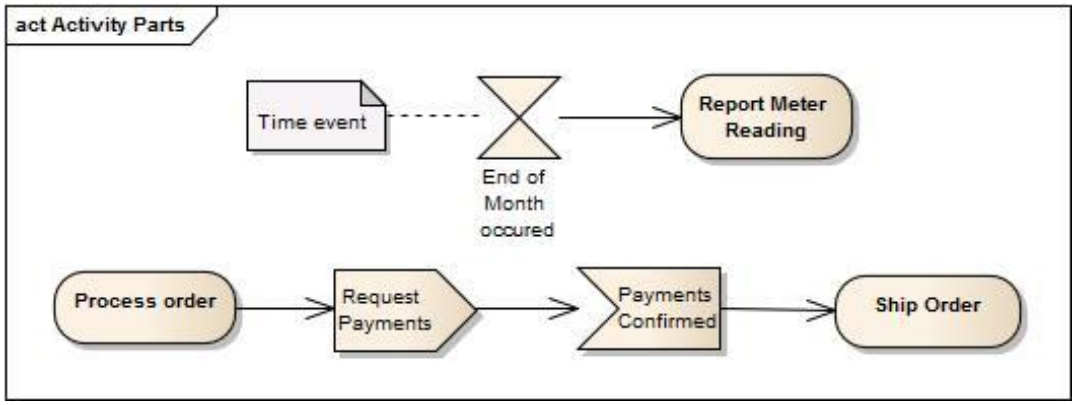

Figure 40 - Activities - Time event - Test data 8

5.4.2 Test cases

5.4.2.1 General Conditions

The mapping between source (UML metamodel) and target (Enterprise Architect database schema) metamodel is defined for all elements of the UML metamodel subset state machine. Elements, properties or associations that were not mapable (metamodel mapping problems) are marked as special case that throws an exception. The execution time of the OCL expression is noticed on the bottom of the status column in milliseconds (rounded up).

5.4.2.2 Connection Point Reference (CPR)

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	N	The entry pseudostates must be pseudostates with kind entryPoint.	CPR Entry ID: 231 Pseudostate IDs: 110, 231, 247	Context ConnectionPointReference inv TUVienna-OclCatalouge: self.entry->notEmpty() implies self.entry->forAll(e:Pseudostate | e.kind = #entryPoint)	True	True	OK Time: 606ms
2	N	The exit pseudostates must be pseudostates with kind exitPoint.	CPR Exit IDs: 232, 235 Pseudostate IDs: 111, 232, 235, 248	Context ConnectionPointReference inv TUVienna-OclCatalouge: self.exit->notEmpty() implies self.exit->forAll(e:Pseudostate | e.kind = #exitPoint)	True	True	OK Time: 605ms

Table 15 - Connection Point Reference test cases

5.4.2.3 Final state

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	N	A final state cannot have any outgoing transitions.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.outgoing->size() = 0	True	True	OK Time: 702ms
2	N	A final state cannot have regions.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.region->size() = 0	True	True	OK Time: 85ms
3	N	A final state cannot reference a submachine.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.submachine->isEmpty()	True	True	OK Time: 80ms
4	N	A final state has no entry behaviour.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.entry->isEmpty()	True	True	OK Time: 76ms
5	N	A final state has no exit behaviour.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.exit->isEmpty()	True	True	OK Time: 74ms
6	N	A final state has no state (do activity) behaviour.	Final state IDs: 182, 188, 195, 219, 241, 249	Context FinalState inv TUViennaOclCatalouge: self.doActivity->isEmpty()	True	True	OK Time: 74ms

Table 16 - Final state test cases

5.4.2.4 ProtocolStateMachine

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	S	A region can have at most one initial vertex.	1 protocol state machine is stored	Context ProtocolStateMachine inv TUViennaOclCatalouge: (not (self.context->isEmpty())) and self.specification->isEmpty()	Exception, because the association self.context navigates to another subset of the UML metamodel	InvalidNavigationNameException, the associations are not part of the statemachine mapping, which are used to navigate to other subsets of the UML metamodel	OK 3) Time: 74ms
2	S	All transitions of a protocol state machine must be protocol transitions. (transitions as extended by the ProtocolStateMachines package).	1 protocol state machine is stored	Context ProtocolStateMachine inv TUViennaOclCatalouge: self.region->forAll(r : Region | r.transition->forAll(t : Transition | t.oclIsTypeOf(ProtocolTransition)))	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.region is not mapable.	OK 1) Time: 72ms
3	S	The states of a protocol state machine cannot have entry, exit, or do activity actions.	1 protocol state machine is stored	Context ProtocolStateMachine inv TUViennaOclCatalouge: self.region->forAll(r :Region | r.subvertex->forAll(v : Vertex | v.oclIsKindOf(State) implies (v.entry->isEmpty() and v.exit->isEmpty() and v.doActivity->isEmpty())))	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.subvertex is not mapable.	OK 1) Time: 76ms
4	S	Protocol state machines cannot have deep or shallow history pseudostates.	1 protocol state machine is stored	Context ProtocolStateMachine inv TUViennaOclCatalouge: self.region->forAll (r : Region | r.subvertex->forAll (v : Vertex | v.oclIsKindOf(Psuedostate) implies((v.kind <> #deepHistory) and (v.kind <> #shallowHistory)))))	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.subvertex is not mapable.	OK 1) Time: 90ms

Table 17 - ProtocolStateMachine test cases

Only few elements of the subset protocol state machine of the UML metamodel state machine package, like the *ProtocolStateMachine* is described in the EA metamodel. Other components like the *ProtocolState* or the *ProtocolTransition* are deduced from the EA API due to a relation to a *ProtocolStateMachine* object. The metamodel mapping (see Chapter 4.2.1.1) concentrates on rules that connects on data stored in the source and the target metamodel and not in data delivered from deduction routines.

5.4.2.5 Pseudostate

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	N	An initial vertex can have at most one outgoing transition.	19 initial pseudostates	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #initial) implies (self.outgoing->size() <= 1)	True	True	OK Time: 398ms
2	N	History vertices can have at most one outgoing transition.	Deep History ID: 304 Shallow History ID: 305	Context Pseudostate inv TUViennaOclCatalouge: ((self.kind = #deepHistory) or (self.kind = #shallowHistory)) implies (self.outgoing->size() <= 1)	True	True	OK Time: 611ms
3	N	In a complete state machine, a join vertex must have at least two incoming transitions and exactly one outgoing transition.	Join ID: 303	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #join) implies ((self.outgoing->size() = 1) and (self.incoming->size() >= 2))	True	True	OK Time: 601ms
4	S	All transitions incoming a join vertex must originate in different regions of an orthogonal state.	Join ID: 303	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #join) implies self.incoming->forAll(t1, t2 : Transition | (t1 <> t2) implies (self.stateMachine.LCA(t1.source, t2.source).container.isOrthogonal = true))	NotSupportedException LCA feature not supported	EA does not support the mapping between Region and Transition	OK 1) Time: 341ms
5	N	In a complete state machine, a junction vertex must have at least one incoming and one outgoing transition.	Fork ID: 302	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #fork) implies ((self.incoming->size() = 1) and (self.outgoing->size() >= 2))	True	True	OK Time: 623ms

6	N	All transitions outgoing a fork vertex must target states in different regions of an orthogonal state.	Fork ID: 302	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #fork) implies self.outgoing->forAll(t1, t2 : Transition | (t1 <> t2) implies (self.stateMachine.LCA(t1.target, t2.target).container.isOrthogonal = true))	NotSupportedException LCA feature not supported	EA does not support the mapping between Region and Transition	OK 1) Time: 623ms
7	N	In a complete state machine, a junction vertex must have at least one incoming and one outgoing transition.	Junction ID: 301	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #junction) implies ((self.incoming->size() >= 1) and (self.outgoing->size() >= 1))	True	True	OK Time: 610ms
8	N	In a complete state machine, a choice vertex must have at least one incoming and one outgoing transition.	Choice ID: 242	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #choice) implies ((self.incoming->size() >= 1) and (self.outgoing->size() >= 1))	True	True	OK Time: 607ms
9	N	The outgoing transition from an initial vertex may have behaviour, but not a trigger or guard.	19 initial pseudostates	Context Pseudostate inv TUViennaOclCatalouge: (self.kind = #initial) implies (self.outgoing.guard->isEmpty() and self.outgoing.trigger->isEmpty())	False	False	OK Time: 660ms

Table 18 - Pseudostate test cases

5.4.2.6 Region

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	S	A region can have at most one initial vertex.	11 regions are stored	Context Region inv TUViennaOclCatalouge: self.subvertex->select(v : Vertex | v.oclIsKindOf(Pseudostate))->select(p : Pseudostate | p.kind = #initial)->size() <= 1	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.subvertex is not mapable.	OK 1) Time: 229ms
2	S	A region can have at most one deep history vertex.	11 regions are stored	Context Region inv TUViennaOclCatalouge: self.subvertex->select (v :Vertex | v.oclIsKindOf(Pseudostate))->select(p : Pseudostate | p.kind = #deepHistory)->size() <= 1	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.subvertex is not mapable.	OK 1) Time: 72ms
3	S	A region can have at most one shallow history vertex.	11 regions are stored	Context Region inv TUViennaOclCatalouge: self.subvertex->select (v :Vertex | v.oclIsKindOf(Pseudostate))->select(p : Pseudostate | p.kind = #shallowHistory)->size() <= 1	Exception, because the EA does not support regions as objects in the database.	InvalidNavigationNameException, because the navigation self.subvertex is not mapable.	OK 1) Time: 68ms

4	N	If a Region is owned by a state machine, then it cannot also be owned by a State and vice versa.	State IDs with regions: 129, 130, 131, 137, 148, 153, 154, 162, 163, 185, 289 No statemachine with a region	Context Region inv TUViennaOclCatalouge: (self.stateMachine->notEmpty() implies self.state->isEmpty()) and (self.state->notEmpty() implies self.stateMachine->isEmpty())	True	True	OK Time: 624ms

5	S	The redefinition context of a region is the nearest containing state machine.	-	Context Region inv TUViennaOclCatalouge:
				redefinitionContext = let sm = containingStateMachine() in
				if sm.context->isEmpty() or sm.general->notEmpty() then
				sm
				else
				sm.context
				endif
					Exception, because let- and if - expressions are not part of the prototype implementation	NotSupportedException, because the let- and if-expressions are not part of the prototype	OK
							2)
							
							Time: 396ms

Table 19 - Region test cases

Regions in the EA metamodel are set as strings in a single column. It is not possible to identify an explicit region in a column using automatic generated SQL commands. The impossible metamodel mapping of UML regions in the EA metamodel leads to several exceptions in the test cases.

5.4.2.7 State

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	F	Only submachine states can have connection point references (CPR).	Submachine state ID: 181, 326 CPR IDs: 231, 232, 235	Context State inv TUViennaOclCatalouge: (self.isSubmachineState = true) implies (self.connection->notEmpty())	False	False No CPR is added to the submachine state manually	OK Time: 522ms
2	F	The connection point references used as destinations/sources of transitions associated with a submachine state must be defined as entry/exit points in the submachine state machine.	Submachine state ID: 181, 326 CPR IDs: 231, 232, 235	Context State inv TUViennaOclCatalouge: self.isSubmachineState = true implies self.connection->forAll (cp : ConnectionPointReference | cp.entry->forAll(p : Pseudostate | p.stateMachine = self.submachine) and cp.exit->forAll (p : Pseudostate| p.stateMachine = self.submachine)))	False	False No CPR is added to the submachine state manually	OK Time: 1sec 349s
3	F	A composite state is not allowed to have both a submachine and regions.	Composite IDs: 178, 181, 185, 226, 246, 289 Submachine state IDs: 181	Context State inv TUViennaOclCatalouge: (self.isComposite = true) implies (not (self.isSubmachineState = true)	False	False State with ID 181 is declared as composite and as submachine state.	OK Time: 383ms
4	N	A simple state is a state without any regions.	42 simple states	Context State inv TUViennaOclCatalouge: (self.isSimple = true) implies (self.region->isEmpty())	True	True	OK Time: 388ms
5	N	A composite state is a state with at least one region.	Composite state IDs: 178, 181, 185, 226, 246, 289	Context State inv TUViennaOclCatalouge: (self.isComposite = true) implies (self.region->notEmpty())	True	False Because the EA metamodel does not support regions, the ocl expression leads to false.	NOK Time: 375ms

6	N	An orthogonal state is a composite state with at least 2 regions.	Orthogonal state ID: 129, 130, 131, 137, 153, 154, 162, 163, 185, 289	Context State inv TUVienna-OclCatalouge: (self.isOrthogonal true) implies (self.region->size() > 1)	True	False Because the EA metamodel does not support regions as objects in the database, only one data row is found. In a data column all regions are defined. The mapping rules of the prototype do not support the identification.	NOK Time: 389ms
7	F	Only submachine states can have a reference state machine.	Submachine state ID: 181, 326	Context State inv TUVienna-OclCatalouge: (self.isSubmachineState = true) implies self.submachine->notEmpty()	False	False Submachine state with ID 181 is not a submachine (declared as instance in EA).	OK Time: 381ms

8	S	The redefinition context of a state is the nearest containing state machine.	-	Context State inv TUVienna-OclCatalouge: redefinitionContext = let sm = containingStateMachine() in
				if sm.context->isEmpty() or sm.general->notEmpty() then
				sm
				else
				sm.context
				endif
					Exception, because let- and if -expressions are not part of the prototype implementation	NotSupportedException, because the let- and if-expressions are not part of the prototype	OK
							2)
							
							Time: 439ms

9	F	Only composite states can have entry or exit pseudostates defined.	Composite state IDs: 178, 181, 185, 226, 246, 289	Context State inv TUViennaOclCatalouge: (self.connectionPoint->notEmpty()) implies (self.isComposite = true)	False	False Composite state with ID 181 has no connectionPoint.	OK Time: 378ms
10	N	Only entry or exit pseudostates can serve as connection points.	Entry point IDs: 110, 231, 247 Exit point IDs: 111, 232, 235, 248	Context State inv TUViennaOclCatalouge: self.connectionPoint->forAll(cp : Pseudostate |cp.kind = #entryPoint or cp.kind = #exitPoint)	True	True	OK Time: 1sec 54ms

Table 20 - State test cases

The UML Specification [**1**], [**2**] declares the properties of a state are as deduced. In the case of the Enterprise Architect these properties are set to false as default. The deduction routines are part of the EA API and do not chance the EA database entries. For comprehensive result the properties are set manually, otherwise all state test cases in Table 20 that includes property references leads to invalid results.

5.4.2.8 State machine

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	S	The classifier context of a state machine cannot be an interface.	4 state machines are stored	Context StateMachine inv TUViennaOclCatalouge: self.context->notEmpty() implies not self.context.oclIsKindOf(Interface)	Exception, because the association self.context navigates to another subset of the UML metamodel	InvalidNavigationNameException The associations are not part of the state machine mapping, which are used to navigate to other subsets of the UML metamodel	OK 3) Time: 391ms
2	S	The context classifier of the method state machine of a behavioural feature must be the classifier that owns the behavioural feature.	4 state machines are stored	Context StateMachine inv TUViennaOclCatalouge: self.specification->notEmpty() implies (self.context->notEmpty() and self.specification.featuringClassifier->exists(c | c = self.context))	Exception, because the associations self.context and self.specification navigates to another subset of the UML metamodel	InvalidNavigationNameException The associations are not part of the state machine mapping, which are used to navigate to other subsets of the UML metamodel	OK 3) Time: 119ms
3	N	The connection points of a state machine are pseudostates of kind entry point or exit point.	8 state machines are stored	Context StateMachine inv TUViennaOclCatalouge: self.connectionPoint->forAll(c : Pseudostate | c.kind = #entryPoint or c.kind = #exitPoint)	True	True	OK Time: 662ms
4	S	A state machine as the method for a behavioural feature cannot have entry/exit connection points.	4 state machines are stored	Context StateMachine inv TUViennaOclCatalouge: self.specification->notEmpty() implies self.connectionPoint->isEmpty()	Exception, because the associations self.context and self.specification navigates to another subset of the UML metamodel	InvalidNavigationNameException The associations are not part of the state machine mapping, which are used to navigate to other subsets of the UML metamodel	OK 3) Time: 73ms

Table 21 - State machine test cases

5.4.2.9 Transition

Nr.	Type	Description	Pre-Condition	Input – OCL Expression	Expected Result	Effective Result	Status
1	N	A fork segment must not have guards or triggers.	Outgoing transitions from a fork segment: 159, 160	Context Transition inv TUViennaOclCatalouge: (self.source.oclIsKindOf(Pseudostate) and self.source.kind = #fork) implies (self.guard->isEmpty() and self.trigger->isEmpty())	True	True	OK Time: 987ms
2	N	A join segment must not have guards or triggers.	Incoming transitions from a join segment: 161, 162, 163	Context Transition inv TUViennaOclCatalouge: ((self.target.oclIsKindOf(Pseudostate)) and (self.target.kind = #join)) implies ((self.guard->isEmpty() and (self.trigger->isEmpty()))	True	True	OK Time: 990ms
3	N	A fork segment must always target a state.	Outgoing transitions from a fork segment: 159, 160	Context Transition inv TUViennaOclCatalouge: (self.source.oclIsKindOf(Pseudostate) and self.source.kind = #fork) implies (self.target.oclIsKindOf(State))	True	True	OK Time: 596ms
4	N	A join segment must always originate from a state.	Incoming transitions from a join segment: 161, 162, 163	Context Transition inv TUViennaOclCatalouge: (self.target.oclIsKindOf(Pseudostate) and self.target.kind = #join) implies (self.source.oclIsKindOf(State))	True	True	OK Time: 616ms
5	F	Transitions outgoing pseudostates may not have a trigger (except for those coming out of the initial pseudostate).	11 transitions outgoing pseudostates (not initial ones) stored.	Context Transition inv TUViennaOclCatalouge: (self.source.oclIsKindOf(Pseudostate) and (self.source.kind <> #initial)) implies self.trigger->isEmpty()	False	False Transitions with ID: 103, 104, 109, 110 have triggers. See account process example.	OK Time: 703ms

6	S	An initial transition at the topmost level (region of a state machine) either has no trigger nor it has a trigger with the stereotype “create.”	85 transitions, 18 initial pseudostate and 35 trigger are stored	Context Transition inv TUViennaOclCatalouge: self.source.oclIsKindOf(Pseudostate) implies (self.source.oclAsTypeOf(Pseudostate).kind = #initial) implies (self.source.container = self.stateMachine.top) implies ((self.trigger->isEmpty()) or (self.trigger.stereotype.name = 'create'))	Exception, because the EA does not support regions as objects in the database. Exception, because the association trigger.stereotype is not part of the state machine subset	InvalidNavigationNameException The associations are not part of the state machine mapping, which are used to navigate to other subsets of the UML metamodel	OK 1) Time: 408ms

7	S	The redefinition context of a transition is the nearest containing state machine.	85 transitions are stored	Context Transition inv TUViennaOclCatalouge: redefinitionContext =
				let sm = containingStateMachine() in
				if sm.context->isEmpty() or sm.general->notEmpty() then
				sm
				else
				sm.context
				endif
					Exception, because let- and if - expressions are not part of the prototype implementation	NotSupportedException, because the let- and if-expressions are not part of the prototype	OK
							2)
							
							Time: 336ms

8	N	A transition with kind local must have a composite state or an entry point as its source.	Transition IDs: 101, 143	Context Transition inv TUViennaOclCatalouge: (self.kind = #local) implies ((self.source.oclIsKindOf(State) and self.source.oclAsType(State).isComposite = true) or (self.source.oclIsKindOf(Pseudostate) and self.source.oclAsType(Pseudostate).kind = #entryPoint))	False	False Only simple states are used by transition with ID: 143. Others are defined right.	OK Time: 1sec 536ms

9	N	A transition with kind external can source any vertex except entry points.	Transition IDs: 99, 145	Context Transition inv TUViennaOclCatalouge: (self.kind = #external) implies (not (self.source.oclIsKindOf(Pseudostate) and self.source.oclAsType(Pseudostate).kind = #entryPoint))	True	True	OK Time: 890ms
10	N	A transition with kind internal must have a state as its source, and its source and target must be equal.	Transition IDs: 100, 144	Context Transition inv TUViennaOclCatalouge: (self.kind = #internal) implies (self.source.oclIsKindOf(State) and self.source = self.target)	True	True	OK Time: 714ms

Table 22 - Transition test cases

5.4.3 Test results

The consistency analysis structures the status of the test cases within a traffic light concept (see Table 23). It reveals that 32 of the 50 OCL expressions of the state machine package are valid – green marked. Only in two cases, the output was wrong, which are marked red. The result sets of the wrong cases contained no model elements, because the EA does not support the UML concept *Region* in the database schema. Regions in the EA are calculated via the position in the visual representation, but it is not possible to refer an instance in the EA database. The yellow state describes the category for failing executions due to deficient support, which is divided in the three subparts mentioned in Chapter 5.4. In nine cases, it is not possible to define a metamodel mapping, because the target metamodel does not support this feature: A region is not a clearly element in the EA database schema. Three out of the sixteen are not supported OCL features (see Table 7, Table 8 and Table 9) and in the last four cases, OCL expressions do not refer to supported subsets.

State	Description	Result
Red	False positive or false negatives	2
Yellow	Failing executions due to deficient support	16
Green	Correct output	32

Table 23 - Consistency analysis - Test results

5.5 Performance analysis

The performance analysis gives an overview, how much time the OCL2SQL transformation in the case of EA takes. Therefore the 50 OCL expressions of the state machine package are executed in a row, based on different sizes of test suites, which are shown in Table 24. For the performance analysis, the test data of the consistency analysis in Chapter 5.4.1 is extended with random test data, consisting of states and transitions.

5.5.1 Test results

Test Suite	Number of model elements	ADOCLE	EA API
A	10-100	8113ms	170382ms
B	100-500	8183ms	445413ms
C	500-5000	9625ms	>1h
D	5000-50000	14262ms	>3h

Table 24 - Performance analysis - Test results

The evaluation revealed that the validation is significantly faster using ADOCLE. The solution, validating models in EA using the COM interface leads to enormous time effort caused by the sequentially testing of the conditions for each element. ADOCLE yields for each test suite a faster validation, as is depicted briefly in the following chart diagram (see Figure 1). The greater the number of elements, the more time is required to perform the validation.

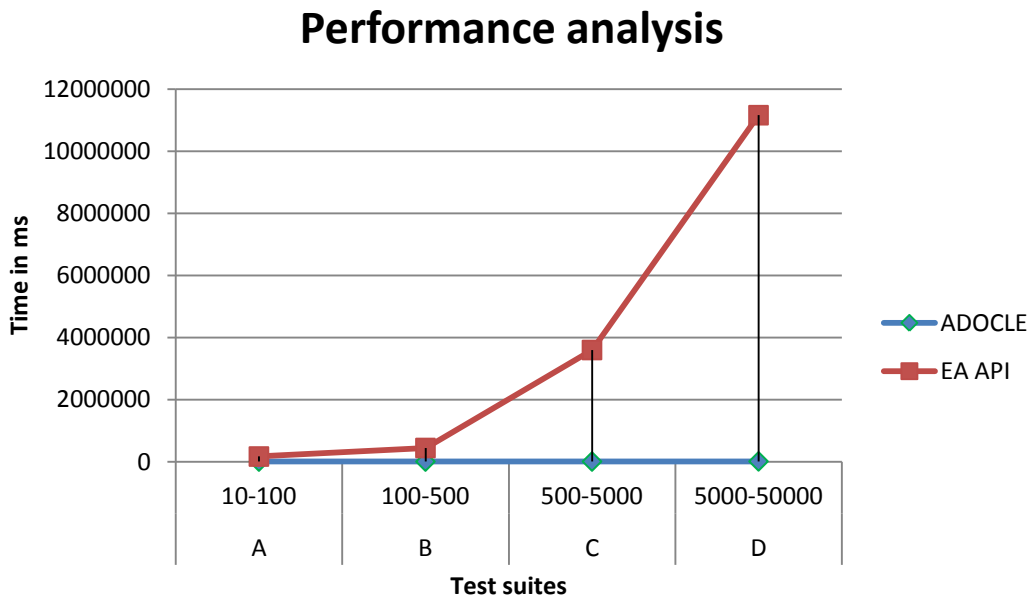

Figure 41 - Performance analysis - Chart diagram

The evaluation shows that the validation could be performed significantly faster using ADOCLE. On the other hand, the consistency analysis figured out a list of problems. It is important that each concept of the source metamodel is represented in the target metamodel. Otherwise, it is not possible to define a metamodel mapping, which leads to transformation and/or execution errors. Additionally, the definition of a metamodel mapping is very time intensive. But the performance of the validation speaks for itself.

6 Related Work

The related work is divided into three parts. First, an overview of similar metamodel-based approaches for OCL compiler and model transformations is given. Second, the main OCL tools and applications are discussed. And third, works concerning verification and validation tools for OCL are presented.

6.1 Overview of metamodel-based OCL compiler

The Object Constraint Language is used as constraint language in the UML Specification to define semantics which cannot be expressed with simple UML class diagrams. But OCL is not limited to UML models. In general, OCL is a constraint language, which can be applied to many kinds of models. Hence, there is a need to use OCL in different ways and on different kind of models at the OMG's modelling layers.

6.1.1 Implementing OCL for multiple Metamodels

In the paper [**45**] the authors discuss an implementation of a compiler of the OCL 2.0 standard for multiple metamodels. The solution provides a bridge to a variety of object-oriented metamodels, represented through a carefully specified set of interfaces to providing a clean and well-defined division between the OCL model and the metamodel to which it is attached. The classes of the bridge package must be supported by any model to interpret and evaluate OCL expression over them. The authors have implemented three bridges for different metamodels: OCL for Java, OCL for Kent Modelling Framework (KMF)[25] and OCL for Eclipse Modelling Framework (EMF). Therefore a mapping of the classes to each metamodel is manually accomplished. The bridge facilitates the use of a shared library, which contains the implementation structure parts shown in Figure 42, in context of a number of different metamodels.

[25] http://www.cs.kent.ac.uk/projects/kmf

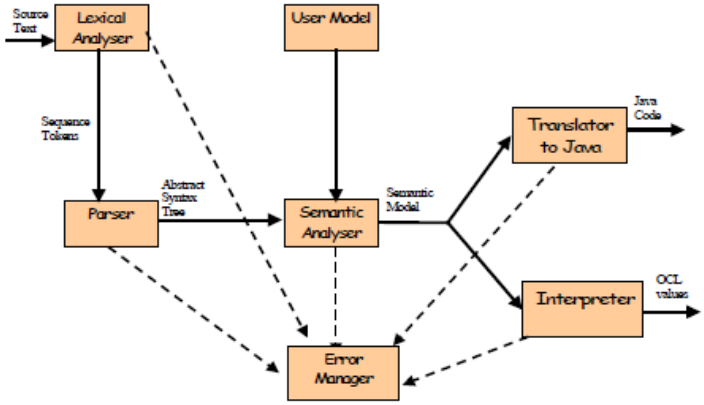

Figure 42 - Implementation structure of the work [45]

The implementation structure follows a typical structure of a translator, consisting of lexical analysis, parsing an abstract syntax tree, semantic analysis and either code generation or evaluation. For the translator, a BNF grammar for a LALR [**51**] parser generator was produced that is suitable for bottom up parser generators like (CUP[26], YACC [**52**], BISON[27]). This grammar was derived from the original OCL standard [**3**] to remove ambiguities in the specification (see Chapter 4.2.2.1). In addition the paper pictures errors and missing parts of the OCL 2.0 specification and suggests options for fixing these problems.

6.1.2 OCL Module in VTMS

Another approach of an OCL compiler, which is based on the .NET platform is discussed in [**53**]. It is an extension of an existing OCL module in the Visual Modelling and Transformation System (VTMS) to support metamodelling and model transformation. VMTS is based on an n-layer environment (composed of autonomous subsystems) to offer graphical metamodelling editing tools. The most interesting component in the VTMS architecture is of course the Constraint Validator Module, which contains the model validation subsystems (see Figure 43) including the OCL compiler, and the related functions.

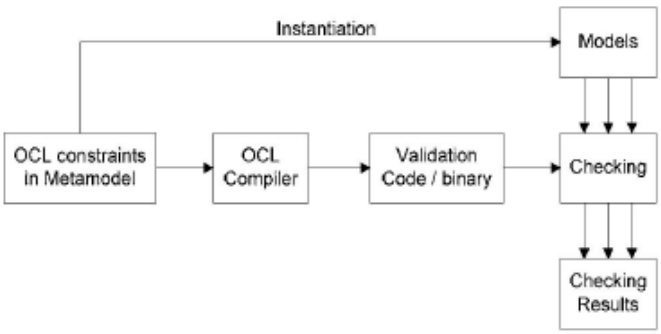

Figure 43 - OCL Module in VTMS

[26] http://www.cs.princeton.edu/~appel/modern/java/CUP/

[27] http://www.gnu.org/software/bison/

The structure of the OCL compiler is similar to a fundamental approach of a compiler. The constraint is defined by the user as OCL expression. The lexical analyser reads the text segment and creates a sequence of tokens, which is performed by a Flex[28.] The syntactical analyser generates a syntax tree, which is reconstructed in a semantic analysed syntax tree after the semantic analysis phase. As next step, this tree is transformed in a CodeDOM tree. CodeDOM is a technology of Microsoft that can describe programs using abstract trees, and it can use this tree representation to generate code to any languages that is supported by the .NET Community. The CodeDOM is used to transform the OCL representation in C# source code to validate a given model. For the support of base types, a class library is also responsible for the type conformance rules (see Chapter 2.2.4) and manages the type checking within the OCL type hierarchy including the OCL types.

In VTMS, models are handled as labelled, directed graphs, which are used in graph transformation algorithms [**54**]. The basic idea of graph transformation is that the current state of a system can be represented as a graph, as well as the state after a computation. Such computations are described by rewriting rules, the atoms of a graph transformation algorithm. A rewriting rule shows the process from one state to another state. Such rules consists of a left part called pattern graph or left-hand-side (LHS) and a right side is called replacement graph or right-hand-side (RHS). Applying a rewrite rule means finding an isomorphic occurrence of the LHS and replacing the graph with the RHS. In VTMS, the LHS and RHS are built from the metamodel elements. This means, an instantiation of LHS in one metamodel can be transformed in an instantiation according to another metamodel. In the paper [**53**] a simple example is described using the same metamodel for the LHS and RHS. But it is also possible to use different metamodels on each side. In [**55**] the graph transformation with VTMS is compared with other graph transformation tools.

6.1.3 Redesign of the Dresden OCL Compiler

The work [**56**] discuss a redesign of the “Dresden OCL” Toolkit to manage basic functionality for processing OCL expression of version 2.0 and the evaluation of well-formedness rules (WFR) on metamodels as well as constraints on models. The work based on previous works of the “Dresden OCL” Team that developed a toolkit to manage OCL expression of version 1.3.

[28] Flex is a tool for generating scanners. See http://flex.sourceforge.net/

The release 1.3 of the “Dresden OCL” Toolkit provided the ability to process preconditions, post conditions and invariants on UML models. Code generators for Java and SQL cover a wide range of applications. Cooperations with partner in the industry leveraged the quality and led to solid base of the “Dresden OCL”.

Due to the requirement to enable the evaluation of WFR on metamodels, a major part of the work is concerned on the alignment of the OCL metamodel to the metamodel of MOF. Typical uses of OCL expressions for UML models, which reside on layer M1, are defined by the OCL metamodel with dependencies to the UML metamodel. Since metamodels on M2 layer like the UML metamodel are instances of the MOF model, but OCL is rather designed to write expression on M1 layer, the evaluation of WFRs on the UML metamodel would not be possible. This is a very important point, because the usage of OCL expressions range from layer M1 to M2. Therefore an adapted OCL metamodel for MOF is necessary. The paper describes the MOF meta-data architecture and the solution of the problem in detail.

In contrast to the already mentioned works [**45**], [**53**], the primary component of the OCL compiler is a MOF repository. According to the paper [**56**], “The purpose of the repository is to manage models and metamodels, to generate particular interfaces for accessing these models, and to provide implementations for the interfaces according to given specifications.” In other words, the provided interfaces of the MOF repository accomplish standard functions like parsing or code generation. The task of the OCL compiler is the same as for the mentioned approaches before and is fulfilled in a similar way.

The code generation of the “Dresden OCL” Toolkit is a transformation of instances of the OCL metamodel to either Java or SQL. Depending on the instances, the output is generated for each OCL metamodel instances sequently. The paper shows an example for transforming an if-expression in OCL to Java source code.

For the evaluation the tool “UML-based Specification Environment”, called USE (see Chapter 6.3.1) is chosen. It can be employed to validate a model, for instance over UML models. It allows defining models on a textual base and associating OCL expression to them. Systems states, so called snapshots can be created to check, if the OCL expressions are valid. Basically, these models can be metamodels and therefore the WFRs can be evaluated. More details about the USE tool can be found in the following section 6.3.1.

6.1.4 Summary

The mentioned works [**45**], [**53**], [**56**] provide an abstract approach of managing OCL expressions for multiple metamodels. The compiler structure is very similar, only few sub components differ in applied technologies, the modified grammar or in a platform specific manner. The works [**45**], [**53**] focus on the usage of OCL expression on user-defined models that are placed on the M1 layer (see Chapter 2.1.1) using the UML modelling diagrams in section 2.1.2. The illustrated example in Figure 6 pictures a class diagram on this layer.

The instances are objects of the defined classes. Examples for the instances and applied OCL expressions are shown in Chapter 3.2.2. The focus of the work in the paper [**56**] is on the different layer of the four-layer metamodel hierarchy shown in Figure 4. The redesign of the "Dresden OCL" Toolkit aims the basic functionality for processing OCL expressions of version 2.0 and the evaluation of well-formedness rules (WFR) defined on metamodels and constraints on normal models (models on M1 layer). In addition the work of the redesign of the "Dresden OCL" Toolkit concentrates on an automatic validation of OCL expression using the UML-based Specification Environment tool. In the following section the tool of the "Dresden OCL" Team is discussed in detail.

6.2 Tool-support

Although still limited, tool support for processing OCL expressions has increased in the last years. This section only deals with the main OCL tools and applications and the features they support, because other papers like [**57**], [**58**] already compared the functionality of OCL tools explicitly. The big players and innovators, "Dresden OCL" and "Eclipse OCL" comprises a summary of its development history, a description of architecture and single components with focus on the OCL compiler, success stories and future work.

6.2.1 Dresden OCL

The "Dresden OCL" is a toolkit that supports the specification and evaluation of OCL expressions for several metamodels and can be used on different metamodel layers. The aim of the "Dresden OCL" is to provide the ability for modelers to integrate OCL tools into their environments and to enable practical experiments with OCL.

In 1999, the development was started with a standard library for processing OCL expressions and a parser for OCL. The work was mainly done by students under coordination of the scientific staff of the Software Technology Group at the technical university of Dresden. The toolkit is still enhanced and maintained mainly by students and scientific staff. The development over the last decade was mainly influenced by the progress in OCL research and the evolving OCL standard. Consequently, the architecture and the components of the toolkit were reengineered in an iterative process. During the decade, there were three major releases implemented. The first release consists of the initially parts implemented in 1999. The second version, called "Dresden OCL2" Toolkit was released in 2005, which contains APIs for the parser and the workbench, code generators for SQL and Java and simple user interfaces, provided as Java libraries. The third version is "Dresden OCL", which is based on the Eclipse SDK and released as a set of Eclipse plug-ins. Nevertheless, a standalone Java library distribution is additionally available. The important innovations of the latest release are a Pivot Model [**47**] and the redesign of the OCL compiler, mentioned in Chapter 6.1.3. These

innovations decouple the OCL parser and interpreter from a specific metamodel and enable connecting the tool to every meta-model based on object-oriented concepts.

"Dresden OCL" is widely used in teaching, research and practise, due to the initially idea of an open-source third-party library for modelling tools. It is also integrated into several CASE tools such as ArgoUML[29] or MagicDraw[30]; provide support for other tools like HOL-OCL[31] and many research projects[32]. ArgoUML and MagicDraw are well-known tools that provide drawing object-oriented models with the UML techniques. HOL-OCL is an interactive proof system for the Object Constraint Language. It is a shallow embedding of OCL in the higher-order logic (HOL) instance of the interactive theorem prover, called Isabelle (see Chapter 6.3.2).

Currently, improvements between the highly extensible parsing and evaluation backend and the frontend are planned. Due to [**59**], there exists a lack of appropriate frontend tooling like advanced OCL editors that adapt to the different application scenarios.

In future, researches for debugging OCL expressions are focussed and scalability enhancement plays a significant role because case studies have shown that there are performance problems evaluating large OCL packages.

6.2.2 Eclipse OCL

The "Eclipse OCL" is an open source project that provides an implementation of the OCL specification for EMF-based and MOF-based models on the Eclipse platform. The core component facilitates APIs for parsing and evaluating OCL constraints and queries, supports processing OCL expression on Ecore and UML models, offers an API for analysing the AST model of OCL expressions and different kind of OCL editors to define OCL expressions.

The initial code contribution (OCL version 1.x) developed by IBM provided a set of APIs for parsing and evaluating OCL expression for Ecore meta-models. The core of the OCL parser is generated by the LALR Parser Generator which is based on an ANTLR grammar. The evolution to OCL 2.x was under auspices of the Eclipse Foundation. A big advantage is that the deployed personnel for the development of the "Eclipse OCL" are also part of the OMG OCL Revision Task Force. Consequently, Eclipse benefits from pioneer solutions for problems in the OCL specification and it can be expected that future changes in the OMG OCL specification are implemented in the "Eclipse OCL".

The evolution led to a transition to a new underlying infrastructure of the "Eclipse OCL" project. The mature code supported Ecore meta-models and evolved in an iterative process to provide UML. Additionally, an OCL console enables interactive experimentation

[29] http://argouml.tigris.org/

[30] http://www.nomagic.com/products/magicdraw.html

[31] http://www.brucker.ch/projects/hol-ocl/index.de.html

[32] http://www.dresden-ocl.org/index.php/DresdenOCL:SuccessStories

with OCL expressions. The dual support for Ecore and UML was a very important fact to remain competitive and was achieved by a shared generic meta-model. An essential part, the evaluator is tightly coupled to Ecore, which makes an independent implementation of the OCL specification hard to achieve. The Pivot metamodel is just a prototype for processing OCL expression of the OCL 2.3.1 specification. It is generated on the fly to provide the OCL functionality for the corresponding metamodel instance (Ecore or UML). The new "Eclipse OCL" exploits Xtext[33] and uses Ecore models via a Pivot models. This enables to use the mature APIs, which offers limited functionality for UML, and the prototypical resolution of OCL expressions. The experimental APIs will be promoted in the Kepler release.

Another important component of the new "Eclipse OCL" is the ImpactAnalyzer which concern with the efficient re-evaluation after a change. By growing models constrained by many OCL invariants in their metamodels, a re-evaluation becomes a performance challenge. Changes to one model can easily affect invariants for model elements in other resources through the freely navigation across resources using OCL expressions. A reliable evaluation after a change mean that all invariants on all context objects have to be validated, what determine in an inefficient scalability. The ImpactAnalyzer exploits the formality of OCL to optimize the re-evaluation and allows a determination of much smaller sets of model elements on which re-evaluation of expressions is necessary.

From the view of an end-user the "Eclipse OCL" provides an interactive support using different kind of editors: An interactive OCL console (including the Essential OCL editor) enables executing queries on models, where an Xtext editor captures embedded OCL statements within an Ecore metamodel. The embedded OCL is executed when invariants are checked, operation bodies executed or property derivations evaluated. The CompleteOCL editor provides the ability to complement a meta-model of an independent documentation. The editor for the OCL standard library is responsible for the definition and the customization of the standard. The "Eclipse OCL" is used in many Eclipse projects such as Eclipse Modelling Framework (EMF) or the Business Intelligence and Reporting Tools[34] (BIRT). Within the OMG context, OCL could be re-used as the foundation for the Model-to-Text (M2T) and the Model-to-Model transformation language, Query/View/Transformation (QVT) [**60**].

In future, the "Eclipse OCL" should deliver solution candidates for the ambiguous and under-specification of the OCL 2.3. Since OCL is used embedded several Eclipse projects, debugging and testing is not easy to provide for OCL. An isolated OCL-oriented debugger is under construction to accompany the Java code generator. Additionally, OCL code generation for embedded OCL expressions in Ecore metamodels for the Java are planned. The release named Kepler may demonstrate the new specifications (UML 2.5 and OCL 2.5), as well as a consistent functionality of all aspects of OCL. The goal for the Kepler release is to support a full model-driven extensibility.

[33] http://www.eclipse.org/Xtext/

[34] http://www.eclipse.org/birt

6.3 Validation/Verification

OCL is a very valuable and expressive language to refine complex models to fulfil the system requirements using invariants, derivation rules, pre and post conditions, etc. Intelligent editors support the model designer to reduce writing mistakes and supports type checking. The mentioned tools in the previous section take care of the syntax of the executed OCL expressions. Nevertheless, syntactic correctness is not enough in model-driven development. If the base already includes errors, the generated components entail them too. The following subsections give an introduction in validating and verifying OCL expressions. These instruments check if the OCL expressions are valid for the model instances of the domain and proof that there exist no inconsistencies or redundancies among them.

6.3.1 UML-based Specification Environment (USE)

The UML-based Specification Environment (USE) enables developers to validate and partly to verify models in the early design phase. In 1998 the approach of USE was published as a Ph. D. project and has been extended by several research publications[35].

A USE specification of a model is a textual description that is similar to the features of UML class diagrams – a notation, most potential users are already familiar with. OCL expressions are used to specify refinements on the model and to evaluate OCL queries. Figure 44 below gives a general view of the USE approach. A very detailed example can be found in [**61**].

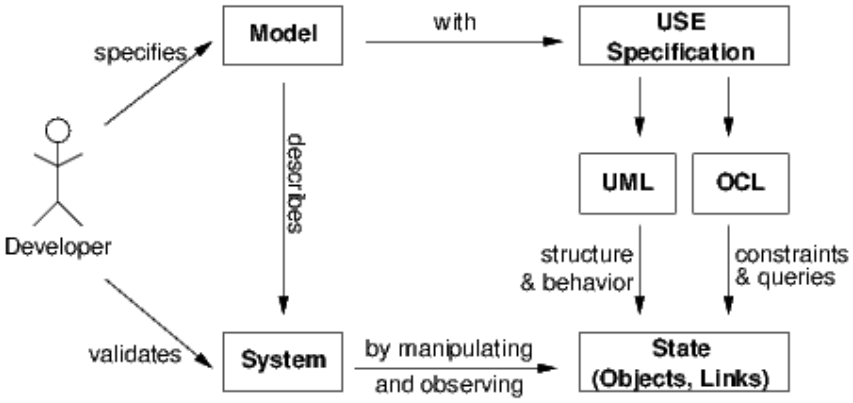

Figure 44 - USE approach

The goal of the USE validation approach is to achieve a valid design before the implementation stage is started. Therefore prototypical instances as a snapshot of a model are generated and compared against the specified model. For each snapshot the specified OCL expressions are automatically checked and the results are visualised by a graphical or textual

[35] http://www.db.informatik.uni-bremen.de/publications/

interface. A result can identify if a model is over-constrained or under-constrained with respect to the design. In the first case: over-constrained, there are snapshots that do not fulfil the OCL expressions. The constraints may be too strong or the design is not adequate and has to be revised. On the other hand, constraints can be too weak and allows undesirable system states. A revise of the constraints has to be done. No matter, which situation occurs the validation only says that the model is correct to the specified snapshots and not that the model is correct at all. The big advantage of the validation is that it is intuitive and can easily be applied by designers without training. It requires a little more effort in the design phase, but it gives the designers immediately feedback and confidence about the analysed model and the snapshots. A snapshot can be built and manipulated manually by using commands of USE, semi-automatically with a graphical built-in snapshot generator or by a Simple OCL-based Imperative Language (SOIL) [**62**]. Every change of a snapshot is recognized and the validation is again automatically executed. These changes can be visualized by sequence diagram.

USE supports debugging OCL expressions with an evaluation browser, which enables the user to analyse the evaluation step by step. The paper [**63**] gives an introduction in combining the approaches of model-driven and test-driven development using the USE tool. The approach proposes a formulation of test suites to improve the model quality by validating constraints with model unit tests. It takes the idea of xUnit[36] test framework and extends assertions with OCL specifics for invariants, pre and post conditions, etc. The graphical user interface to provide an easy access is under construction.

The “UML-based Specification Environment” tool has been utilized in many case studies and modelling research projects in international studies for analysing the results of the research studies. The USE tool also provides the ability of doing conformance tests for CASE tools by analysing the well-formedness rules (WFR) of the UML metamodel. The flexible textual specification of the USE approach allows designing according to the UML standard and validating the output of the CASE tool against the WFR. Integrating USE in CASE tools enables automatic checks of the specified model and immediately feedback and confidence for the designer. Additionally, the WFR do not have to be hard-coded in the CASE tools and allow a faster adaption to evolving standards.

According to the report [**58**], USE will facilitate more UML 2.x features. Further, a redesign of the user interface is panned, as well as the deployment of a small API. In the meantime, the focus is on the integration of SOIL. Apart from these conceptional extensions, continuous integration of smaller changes, improvements and the elimination of bugs are on the agenda.

[36] xUnit is the name for various frameworks for automated software testing.
The best-known representative is JUnit for the programming language Java.

6.3.2 HOL-OCL

The UML techniques in combination with OCL expressions achieve an adequate precision to assure the consistency of specification documents. For example, type-checking and well-formedness rules can be used to identify inconsistencies in the early progress of the design phase. As the previous sections particularized, a-posteriori technique called validation guarantee a correct transition from a specification to its implementation. In other words, the instances are valid against to its model. These techniques are summarized under the term formal methods, where the power of these techniques depends on the degree of precision. Advanced correctness may require a more complete reasoning. But reasoning on OCL is undecidable in general. User interaction to restrict the OCL expression is required in finite space to guarantee termination. Current verification tools provide support to manage the requirements.

Higher-order logic and object constraint language (HOL-OCL) is an interactive theorem-proving environment for UML models annotated with OCL expressions. It is based on the UML and OCL specification and is incorporated in the MDE framework. The HOCL-OCL represents a shallow embedding into the Isabelle theorem, based on the su4sml to ensure the consistency of the formal semantics, in other words, that the constraints are satisfiable. The purpose of HOL-OCL is to prove the satisfiability of OCL expressions on UML models. Several proof-procedures provide a logical framework for object-oriented modelling and reasoning which allows formal derivations establishing the consistency of models.

In the meantime, HOL-OCL only supports UML class diagrams with restrictions of qualifiers of associations ends, enumeration and association classes. There is also a limitation of the data types, only the basic data types of OCL are supported. OclVoid, OCLModelElementType or OclType cannot be modelled explicitly in HOL-OCL.

7 Conclusion and Future Work

7.1 Conclusion

The need for validating models automatically, in the early design phase arose with the emergence of the Model-Driven Development (MDD) paradigm. Models are not any longer used just for brainstorming and communication purposes. Models play a significant role in the development of systems. Thus, the correctness of these models is quite essentially. Sophisticated modelling techniques have been invented to ensure a consistent and comprehensive technology base. The Object Management Group (OMG) suggests the usage of the Unified Modelling Language (UML) to define platform-independent models. UML has been widely accepted as the de facto standard for modelling object-oriented systems. This master thesis pointed out that UML models are not necessary expressive enough to provide all complex aspects of a system. Some aspects of the domain are more expressed in natural language. Practise has shown that such situations will lead to ambiguities and to errors. Therefore the Object Constraint Language (OCL) can be used to specify conditions on models that must hold for the system.

Apart from the development of standards for modelling systems, several vendors developed modelling tools or so called Computer-Aided Software Engineering (CASE) tools to provide a wide range of instruments to support the MDD approach. The support of the sophisticated techniques depends on the vendor's realization of the standards. In general there is a lack of validating models in CASE tools, which is essential in the early design phase. Additionally, there is only little support for writing OCL expressions for models. Thus, models defined by users of such tools are not necessarily valid against the wellformedness rules (WFR) - described with OCL - in the UML specification. It is a huge value for CASE tools to support the definition and validation of OCL expressions to provide valid models.

Based on the current standards and modelling an adaptable approach, called ADOCLE was conducted. The aim of ADOCLE is to manage the transformation of OCL expressions in different environments or CASE tools. Therefore the OCL expressions are applied to a selectable source metamodel, which is mapped to an arbitrary target metamodel. ADOCLE generates a semantically equivalent expression of an OCL expression in the target metamodel depending on transformation patterns. The equivalent target metamodel expression is derived from the mapping between the source and target metamodel.

For the prototypical implementation, the UML metamodel as source metamodel and the well-formedness OCL expressions in the UML specification was used. Enterprise Architect is chosen as a CASE tool, which uses a generic metamodel to provide the support of drawing

models of different kinds. It is expressed as database schema and defines attributes, connectors, elements and operations of specific types depending on the supported kinds of models. Thus, the models are stored in a database. The target metamodel used for this master's thesis is the physical database schema of EA, that is described with Structured Query Language (SQL) and based on the SQL-92 standard. As the ADOCLE approach defines, a metamodel mapping between the UML metamodel and the EA database schema is necessary. Therefore, a metamodel mapping was determined based on patterns, which identifies equivalent UML metamodel elements - the stored models in the database of the EA. An OCL expression applied on the UML metamodel is analysed by an OCL interpreter, which is created by the ANTLR parser generator using the EBNF of the OCL grammar. Thus, each OCL expression is validated against the OCL grammar. The output of the OCL interpreter is an abstract syntax tree (AST) that is used for the translation to an SQL expression. Finally, the generated SQL expression is executed to provide an automatic validation of the given OCL expression. Thus, the well-formedness rules - described with OCL - in the UML specification are used to validate the models in the Enterprise Architect.

The prototype pointed out that it is possible to transform OCL expressions applied on a selectable metamodel to a target metamodel using the concept of ADOCLE and to validate the models of the target metamodel. The advantage of ADOCLE is a platform independent approach providing the functionality of OCL in several environments and CASE tools. The implementation of OCL can be reused for each environment to provide an automatic validation in the early design phase. ADOCLE also supports by generating valid OCL expression using word completion instruments. But a big disadvantage is the huge effort of the metamodel mapping definition between the source and the target metamodel. Additionally, if a metamodel mapping for a necessary element is not possible, ADOCLE leads to transformation and/or execution errors. Thus, ADOCLE is highly influenced by the metamodel mapping.

The consistency analysis reflects the dependency of the metamodel mapping. For the evaluation, the state machine package with its 50 well-formedness rules was chosen, which contains a wide range of modelling paradigms and represents the complexity of the UML specification adequately. The consistency analysis shows that 32 of 50 OCL expressions provide the correct outcome. While in 9 cases, the semantically equivalent SQL expressions cannot be created, because a single element, the UML concept *Region* could not be mapped to the target metamodel. The database schema of the Enterprise Architect does not store the UML concept *Region* in the database as a referable element. That shows the highly influence of the metamodel mapping on the OCL transformation. Other failing issues are caused by the distinction to the state machine package. Extensions that are part of the future work will resolve these issues.

The performance analysis revealed that the validation is significantly faster using ADOCLE. The solution, validating models in EA using the COM interface leads to enormous time effort caused by the sequentially testing of the conditions for each element. ADOCLE yields for each size of test suites a faster validation. The greater the number of elements, the more time is required to perform the validation. Although there is still potential to improve the performance of ADOCLE. These issues are described in the following chapter.

7.2 Future Work

In order to provide comprehensive functionality for validating models in different environments, the prototypical implementation of ADOCLE was built in this master's thesis has to be extended. The following extensions are the most interesting ones:

- **Modelling concepts:** The prototypical implementation of ADOCLE is restricted in the UML modelling concepts very much and only supports state machine. An expansion of the supported modelling concepts constitutes a valuable extension. Therefore, the metamodel mapping has to be extended as well as the included WFR of the modelling concepts has to be checked. An expansion of modelling concepts that are already referred by the OCL expression in the consistency analysis would resolve the failing executions. These issues should be taken into account for the selection of the modelling concepts.
- **Parallel execution:** The support of a parallel transformation of OCL expressions or sub expressions (see Chapter 4.2.3.2) is also a reasonable extension to improve the performance of ADOCLE. Currently the prototype executes sub expressions sequentially and does not incorporate concurrent programming.
- **Graphical editor:** The ADOCLE prototype provides only a list of identifiers for valid and invalid model elements. The existing user interface could be extended with more details about the model elements. Another issue is the integration of ADOCLE in the Enterprise Architect as an Add-In. Invalid models could be colour marked and notes for the incorrectness could be attached

8 List of figures

Figure 1 - MDA approach 1
Figure 2 – Adaptable OCL Engine (ADOCLE) 3
Figure 3 – Approach of ADOCLE prototype mapping between UML and EA 4
Figure 4 - Four-layer metamodel hierarchy 10
Figure 5 - Static aspects of UML 15
Figure 6 - Motivation example - UML class diagram 16
Figure 7 - Association-end and Navigation 20
Figure 8 - Type Hierarchy for OCL 24
Figure 9 - Excerpt of the OCL expression package 26
Figure 10 - Example of expression types 27
Figure 11 - Object diagram for OCL2SQL demonstration 36
Figure 12 - Equal schemas due to the class-to-table mapping approach 41
Figure 13 - Semantically overlapping schemas 42
Figure 14 - Abstract view of ADOCLE 43
Figure 15 - ADOCLE 44
Figure 16 – Metamodel Loader 47
Figure 17 - Metamodel mapping concept 50
Figure 18 - Element mapping 51
Figure 19 - Property mapping 51
Figure 20 - Enumeration mapping 52
Figure 21 - Association mapping 52
Figure 22 – OCL interpreter 54
Figure 23 - Module three - Strategy Pattern 55
Figure 24 - Design Patterns used for the OCL Transformator 56
Figure 25 - OCL constraint basic structure 57
Figure 26 - OCL Parser Output for the example 58
Figure 27 - Behaviour of a Transformation Process for the example 60
Figure 28 - Complex OCL example 61
Figure 29 - Subselect algorithm example 63
Figure 30 - Subselect algorithm SQL 64
Figure 31 - Implies example OCL expression 69
Figure 32 - ADOCLE UI 72
Figure 33 - Cash machine example - Test data 1 76
Figure 34 - Traffic light example - Test data 2 77
Figure 35 - Account process - Test data 3 78
Figure 36 - Protocol state machine - Test data 4 79
Figure 37 - Transition types - Test data 5 79

Figure 38 - State types - Test data 6 80
Figure 39 - Example for special state machine components – Test data 7 81
Figure 40 - Activities - Time event - Test data 8 81
Figure 41 - Performance analysis - Chart diagram 98
Figure 42 - Implementation structure of the work [45] 100
Figure 43 - OCL Module in VTMS 100
Figure 44 - USE approach 106

9 List of tables

Table 1 - OCL basic types 22
Table 2 – OCL Collection types 23
Table 3 - Data entries in table Person 35
Table 4 - Data entries in table Vehicle 35
Table 5 - Data entries in table Company 36
Table 6 - Data entries in table Job 36
Table 7 - List of supported OCL expressions 65
Table 8 - List of supported OCL operators 66
Table 9 - List of supported OCL operations 67
Table 10 - Implication truth table 68
Table 11 - Implies example set A elements 69
Table 12 - Implies example set B1 validation 69
Table 13 - Implies example set B2 validation 69
Table 14 - Statistical hypothesis categories 74
Table 15 - Connection Point Reference test cases 82
Table 16 - Final state test cases 83
Table 17 - ProtocolStateMachine test cases 84
Table 18 - Pseudostate test cases 87
Table 19 - Region test cases 89
Table 20 - State test cases 92
Table 21 - State machine test cases 93
Table 22 - Transition test cases 96
Table 23 - Consistency analysis - Test results 97
Table 24 - Performance analysis - Test results 97

10 List of Mapping Examples

Mapping Example 1 - Attribute 37
Mapping Example 2 - Many-to-One Relation Manager 37
Mapping Example 3 - Many-to-One Relation Vehicle Owner 38
Mapping Example 4 - One-to-Many Relation 38
Mapping Example 5 - Combination of relations and attributes 38
Mapping Example 6 - Operation 39

11 References

[1] Object Management Group. (2006, May) OMG Unified Modelling Language (OMG UML) Infrastructure, Version 2.0. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/

[2] Object Management Group. (2010, May) OMG Unified Modelling Language (OMG UML) Superstructure, Version 2.3. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[3] Object Management Group. (2006, May) Object Constraint Language, Version 2.0. http://www.omg.org/spec/OCL/2.0/PDF/

[4] B. W. Boehm, "Software Engineering Economics," *IEEE Transactions on Software Engineering*, no. 12, pp. 4-21, January 1984.

[5] Sparx System. (2013, January) Enterprise Architect. http://www.sparxsystems.eu/enterprisearchitect/

[6] Eclipse Model Development Tools. (2013, January) Eclipse OCL. http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl

[7] International Organization for Standardization. (1992) Database Language SQL. http://savage.net.au/SQL/sql-92.bnf.html

[8] M. Hitz and G. Kappel, *UML@Work*.: Dpunkt Verlag, 1999.

[9] B. Österreich, *Analyse und Design mit der UML 2.5: Objektorientierte Softwareentwicklung*.: Oldenbourg Wissenschaftsverlag, 2012.

[10] G. Booch, "The Booch method: process and pragmatics," in *Object development methods*. New York: SIGS Publications, Inc., 1994, pp. 149-166.

[11] J. Rumbaugh, Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object Oriented Modeling and Design*. New Jersey, USA: Prentice Hall, 1991.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object Oriented Software Engineering: A Use Case Driven Approach*.: Addison-Wesley, 1992.

[13] International Organization for Standardization. ISO/IEC 14977:1996(E). www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[14] E. Seidewitz, "What Models Mean," *IEEE Software*, no. 20, pp. 26-32, 2003.

[15] R. France, A. Evans, K. Lano, and B. Rumpe, "The UML as a formal modeling notation," *Computer Standards & Interfaces*, vol. 19, no. 7, pp. 325-334, 1998.

[16] C.B. Jones, *Systematic Software Development using VDM*.: In Prentice Hall, 1990.

[17] J. Davies and J. Woodcock, *Using Z: Specification, Refinement and Proof*.: In Prentice Hall, 1996.

[18] S. Schneider, *B Method - An introduction*.: Palgrave, Cornersones of Computing series, 2001.

[19] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*,

1st ed. Bosten, USA: Addison-Wesley, 1998.

[20] M. Gogolla, *An Extended Entity-Relationship Model: Fundamentals and Pragmatics*. Berlin: Springer, 1993.

[21] W. Kim and F. H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications*. Michigan, USA: ACM Press, 1989.

[22] S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*.: Prentice Hall, 1994.

[23] M. Vaziri and D. Jackson, "Some Shortcomings of OCL, the Object Constraint Language of UML," in *TOOLS '00 Proceedings of the Technology of Object-Oriented Languages and Systems*, Washington, 1999, pp. 555-572.

[24] M. Gogolla and U. Hohenstein, "Towards a semantic view of an extended entity-relationship model," *ACM Transactions on Database Systems (TODS)*, no. 16, pp. 369 - 416, September 1991.

[25] M. Gogolla and M. Richters, "On constraints and queries in UML," in *The Unified Modeling Language - Technical Aspects and Applications*. Heidelberg, Deutschland: Physica-Verlag, 1998, pp. 109-121.

[26] R. F. van der Lans, *Introduction to SQL: Mastering the Relational Database Language*, 4th ed.: Pearson, 2006.

[27] K. Kline, Kline D., and B. Hunt, *SQL in a Nutshell*, 3rd ed. O'Reilly Media, USA: O'Reilly Media, 2008.

[28] M. Blaha and W. Premerlani, *Object-Oriented Modeling and Design for Database Applications*, 1st ed. Groningen, The Netherlans: Prentice Hall, 1997.

[29] A. Schmidt, Untersuchungen zur Abbildung von OCL-Ausdrücken auf SQL (Master Thesis), 1998.

[30] B. Demuth and H. Hussmann, "Using UML/OCL Constraints for Relational Database Design," in *UML'99 Proceedings of the 2nd international conference on the unified modeling language*, Heidelberg, 1999.

[31] S. Loecher, B. Demuth, and H. Hussmann, "OCL as a Specification Language for Business Rules in Database Applications," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, London, 2001, pp. 104-117.

[32] A. De Almeida, G. Boff, and J. L. De Oliveira, "A Framework for Modeling, Building and Maintaining Enterprise Information Systems Software," in *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, USA: DC, 2009, pp. 115-125.

[33] C. Calgano, W. Taha, L. Huang, and X. Leroy, "Implementing multi-stage languages using ASTs, Gensym and Rejection," in *Generative Programming and Component Engineering*

(GPCE) Conference, LNCS 2830, 2003, pp. 57-76.

[34] T. Cormen, Ch. Leiserson, R. Rivest, and C. Stein, "Depth-first search," in *Indrocution to Algorithms*, MIT Press and McGraw-Hill, Ed. Cambridge, 2001, ch. 22.3, pp. 540-549.

[35] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys (CSUR) - Special issue on heterogeneous databases*, no. 22, pp. 183-236, September 1990.

[36] R. Blake, "A survey of Schema Matching Research," University of Massachusetts Boston, Boston, UMBCMWP 1031 2007.

[37] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal — The International Journal on Very Large Data Bases*, no. 10, pp. 334-350, December 2001.

[38] H. Wache et al., "Ontology-based integration of information - a survey of existing approaches," *In Proceedings of the workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 108-117, 2001.

[39] R.A. Brooker, I.R. Maccallum, D. Morris, and J.S. Rohl, "The Compiler Compiler," in *Annual Review in Automatic Programming, Vol. 3*, London, 1963, pp. 229-275.

[40] J. Jørgensen, "Generating a compiler for a lazy language by partial evaluation," in *POPL '92 Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, USA, 1993, pp. 258-268.

[41] Ch. Consol and O. Danvy, "Partial evaluation: Principles and Perspectives," in *POPL'93 Proceedings of the ACM-SIGPLAN Symposium on Principiles of Programming Languages*, 1993.

[42] T. Parr, *The Definitive Antlr Reference: Building Domain-Specific Languages*. USA: San Fransico: The Pragmatic Programmers, 2007.

[43] L. M. Garshol. (2008, August) BNF and EBNF: What are they and how do they work. www.garshol.priv.no/download/text/bnf.html

[44] Ch. Ullenboom. (2007, Juli) Das ANTLR Eclipse-Plugin. http://www.tutego.de/blog/javainsel/2007/07/das-antlr-eclipse-plugin/

[45] D. Akehurst and O. Patrascoiu, "OCL 2.0 - Implementing the Standard for Multiple Metamodels," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 102, pp. 21-41, November 2004.

[46] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Amsterdam: Addison-Wesley Longman, 1994.

[47] F. Heidenreich, C. Wende, and B. Demuth, "A Framework for Generating Query Language Code From OCL Invariants," in *Proceeding of the Workshop Ocl4All - Modeling Systems with OCL at Models* , Berlin, 2008.

[48] R. Carnap, *Introduction to Symbolic Logic and Its Applications*. New York, 1958.

[49] E. Mendelson, *Introduction to Mathemetical Logic*, 4th ed., Chapman & Hall, Ed. London: Springer, 1997.

[50] A. Tarski, *On the Concept of Logical Consequence*. Indianapolis: Hackett, 1983, pp. 409-420.

[51] F. L. Deremer, "PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES," Massachusetts Institute of Technology , USA, Dissertation 1969.

[52] S. C. Johnson, "Yacc: Yet Another Compiler Compiler," Bell Laboratories, New Jersey, Technical Report 1975.

[53] G. Mezei, L. Lengyel, T. Levendovszky, and H. Charaf, "Extending an OCL Compiler for Metamodeling and Model Transformation Systems: Unifying the Twofold Functionality ," in *Intelligent Engineering Systems - INES '06. Proceedings*, Budapest, 2006, pp. 57-62.

[54] R. Heckel, "Graph Transformation in a Nutshell," *Electronic Notes in Theoretical Computer Sciene*, vol. 148, 2006.

[55] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, and T. Levendovszky, "Model transformation by graph transformation: A comparative study," *MTiP: International Workshop on Model Transformations in Practise (Satellite Event of MoDELS)*, 2005.

[56] S. Loecher and S. Ocke, "A Metamodel-Based OCL-Compiler for UML and MOF," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 102, pp. 43-61, November 2004.

[57] J. Cabot and E. Teniente, "Constraint Support in MDA Tools: A Survey," in *ECMDA-FA'06 Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, Heidelberg, 2006, pp. 256-267.

[58] J. D. Chimiak-Opoka et al., "OCL Tools Report based on the IDE4OCL Feature Model," *Electronic Communications of the EASST*, vol. 44, 2011.

[59] F. Heidenreich et al., "Integrating OCL and textual modelling languages," in *MODELS'10 Proceedings of the 2010 international conference on Models in software engineering*, Heidelberg, 2010, pp. 349-363.

[60] Object Management Group. (2011, January) Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.1/PDF/

[61] M. Richters. (2013, January) The UML-based Specification Environment. sourceforge.net/apps/mediawiki/useocl/index.php

[62] F. Büttner and M. Gogolla, "Modular Embedding of the Object Constraint Language," in *SBMF'11 Proceedings of the 14th Brazilian conference on Formal Methods: Foundations and Applications*, Heidelberg, 2011, pp. 124-139.

[63] L. Hamann and M. Gogolla, "Improving Model Quality by Validating Constraints with

Model Unit Tests," in *MODEVVA '10 Proceedings of the 2010 Workshop on Model-Driven Engineering, Verification, and Validation*, Washington, 2010, pp. 49-54.

[64] A.J. Lait and D. Randell, "An Assessment of Name Matching Algorithms," University of Newcastle, 1996.

[65] B.M. Diaz, "Computers in Genealogy," in *Nominal data visualisation: The Star-Trek Paradigm*., 1994, pp. 23-34.

[66] Software Technology Group at Technische Universität Dresden. (1999) Dresden OCL. http://dresden-ocl.sourceforge.net/

[67] M. Richters and M. Gogolla, "Validating UML Modells and OCL Constraints," in *UML'00 Proceedings of the 3rd international conference on the unified modeling language: advancing the standard*, Heidelberg, 2000.

[68] E. Song, R. B. France, D. K. Kim, and S. Gosh, "Using Roles for Pattern-based Model Refactoring," in *Proceedings of the Workshop on Critical Systems Development with UML (CSDUML '02)*, 2002.

[69] D. H. Akehurst and Kent S., "A Relational Approach to Defining Transformations in a Metamodel," in *UML '02 Proceedings of the 5th International Conference on The Unified Modeling Language* , London, 2002, pp. 243-258.

[70] F. Jounalt and I. Kurtev, "On the architectural alignment of ATL and QVT," in *SAC '06 Proceedings of the 2006 ACM symposium on Applied computing*, New York, 2006, pp. 1188-1195.

[71] S. J. Miller, "Pattern-based model approach: A metamodel-based approach to model evolution," Southern University, Baton Rouge, Dissertation 2004.

[72] G. Bouchard and Ch. Pouyez, "Historial Methods," in *Name Variations And Computerised Record Linkage*.: Spring, 1980, pp. 119-125.

[73] J. M. Kim. (1995) Name Matching for Data Quality Mediator.

[74] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*. Amsterdam: Addison-Wesley Professional, 2008.